Университет ИТМО
Кафедра ВТ

Языки системного программирования

Лабораторная работа №5

Группа P3210
Нгу Фыонг Ань
ПРОВЕРИЛ:
Лукьянов Николай Михайлович

2018 год

13.10 Assignment: Image Rotation

You have to create a program to rotate a BMP image of any resolution to 90 degrees clockwise.

#Code

#main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "bitmap.h"

int main(void){
   char *error = NULL;
   BMPImage *image = read_image("in3.bmp", &error);
   BMPImage *out = rotateRight(image, &error);
   write_image("out.bmp", out, &error);
   return EXIT_SUCCESS;
}

BMPImage *read_image(const char *filename, char **error){
   FILE *input_ptr = _open_file(filename, "rb");

   BMPImage *image = read_bmp(input_ptr, error);
   if (*error != NULL)    {
      _handle_error(error, input_ptr, image);
   }
   fclose(input_ptr);
   return image;
}

void write_image(const char *filename, BMPImage *image, char **error){
   FILE *output_ptr = _open_file(filename, "wb");
   if (!write_bmp(output_ptr, image, error))    {
      _handle_error(error, output_ptr, image);
   }
   fclose(output_ptr);
}
```

```
FILE *_open_file(const char *filename, const char *mode){
   FILE *fp = fopen(filename, mode);
   if (fp == NULL)    {
      fprintf(stderr, "Could not open file %s", filename);
      exit(EXIT_SUCCESS);
   }
   return fp;
}

void _handle_error(char **error, FILE *fp, BMPImage *image){
   fprintf(stderr, "ERROR: %s\n", *error);
   _clean_up(fp, image, error);
   exit(EXIT_SUCCESS);
}
void _clean_up(FILE *fp, BMPImage *image, char **error){
   if (fp != NULL)    {
      fclose(fp);
   }
   free_bmp(image);
   free(*error);
}
```

#bitmap.c

```
#include "bitmap.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAGIC_VALUE        0x4D42
#define NUM_PLANE          1
#define COMPRESSION        0
#define NUM_COLORS         0
#define IMPORTANT_COLORS   0
#define BITS_PER_PIXEL     24
#define BITS_PER_BYTE      8
```

```
BMPImage *read_bmp(FILE *fp, char **error){
   BMPImage *image = malloc(sizeof(*image));
   if (!_check(image != NULL, error, "Not enough memory"))    {
      return NULL;
   }
   // Read header
   rewind(fp);
   int num_read = fread(&image->header, sizeof(image->header), 1, fp);
   if(!_check(num_read == 1, error, "Cannot read header"))    {
      return NULL;
   }
   // Check header
   bool is_valid_header = check_bmp_header(&image->header, fp);
   if(!_check(is_valid_header, error, "Invalid BMP file"))    {
      return NULL;
   }
   // Allocate memory for image data
   image->data = malloc(sizeof(*image->data) * image->header.image_size_bytes);
   if (!_check(image->data != NULL, error, "Not enough memory"))    {
      return NULL;
   }
   // Read image data
   num_read = fread(image->data, image->header.image_size_bytes, 1, fp);
   if (!_check(num_read == 1, error, "Cannot read image"))    {
      return NULL;
   }
   return image;
}

bool write_bmp(FILE *fp, BMPImage *image, char **error){
   // Write header
   rewind(fp);
   int num_read = fwrite(&image->header, sizeof(image->header), 1, fp);
   if (!_check(num_read == 1, error, "Cannot write image"))    {
      return false;
```

```
    }
    // Write image data
    num_read = fwrite(image->data, image->header.image_size_bytes, 1, fp);
    if (!_check(num_read == 1, error, "Cannot write image"))    {
        return false;
    }
    return true;
}

bool check_bmp_header(BMPHeader* bmp_header, FILE* fp)
{
    return
        bmp_header->type == MAGIC_VALUE
        && bmp_header->offset == BMP_HEADER_SIZE
        && bmp_header->dib_header_size == DIB_HEADER_SIZE
        && bmp_header->num_planes == NUM_PLANE
        && bmp_header->compression == COMPRESSION
        && bmp_header->num_colors == NUM_COLORS && bmp_header->important_colors == IMPORTANT_COLORS
        && bmp_header->bits_per_pixel == BITS_PER_PIXEL
        && bmp_header->size == _get_file_size(fp) && bmp_header->image_size_bytes ==
_get_image_size_bytes(bmp_header);
}

void free_bmp(BMPImage *image){
    free(image->data);
    free(image);
}

BMPImage *rotateRight(BMPImage *image, char **error) {
    BMPImage *new_image = malloc(sizeof (*new_image));
    int y = 0;
    int x = image->header.width_px;
    int w = image->header.height_px;
    int h = image->header.width_px;

    // Update new_image header
```

```
new_image->header = image->header;
new_image->header.width_px = image->header.height_px;
new_image->header.height_px = image->header.width_px;
new_image->header.image_size_bytes = _get_image_size_bytes(&new_image->header);
new_image->header.size = BMP_HEADER_SIZE + new_image->header.image_size_bytes;

// Allocate memory for image data
new_image->data = malloc(sizeof (*new_image->data) * new_image->header.image_size_bytes);
if (!_check(new_image->data != NULL, error, "Not enough memory")) {
   return NULL;
}
int position_y = y * _get_image_row_size_bytes(&image->header);
int position_x_row = _get_position_on_row(x, &image->header)-3;
int new_index = 0;

for (int i = 0; i < h; i++) {
   for (int j = 0; j < w; j++) {
      // Iterate image's pixels
      for (int k = 0; k < 3; k++) {
            new_image->data[new_index] = image->data[position_y + position_x_row];
            new_index++;
            position_x_row++;
      }
      position_y += _get_image_row_size_bytes(&image->header);
      position_x_row = _get_position_on_row(x, &image->header)-3;
   }
   // Add padding to new_image
   int padding = _get_padding(&new_image->header);
   for (int l = 0; l < padding; l++) {
      new_image->data[new_index] = 0x00;
      new_index++;
   }
   position_y = y * _get_image_row_size_bytes(&image->header);
   x--;
   position_x_row = _get_position_on_row(x, &image->header)-3;
}
```

```
    return new_image;
}
long _get_file_size(FILE *fp){
    // Get current file position
    long current_position = ftell(fp);
    if (current_position == -1)    {
        return -1;
    }
    // Set file position to the end
    if (fseek(fp, 0, SEEK_END) != 0)    {
        return -2;
    }
    // Get current file position (now at the end)
    long file_size = ftell(fp);
    if (file_size == -1)    {
        return -3;
    }
    // Restore previous file position
    if (fseek(fp, current_position, SEEK_SET) != 0)    {
        return -4;
    }
    return file_size;
}

int _get_image_size_bytes(BMPHeader *bmp_header){
    return _get_image_row_size_bytes(bmp_header) * bmp_header->height_px;
}

int _get_image_row_size_bytes(BMPHeader *bmp_header){
    int bytes_per_row_without_padding = bmp_header->width_px * _get_bytes_per_pixel(bmp_header);
    return bytes_per_row_without_padding + _get_padding(bmp_header);
}

int _get_padding(BMPHeader *bmp_header){
    return (4 - (bmp_header->width_px * _get_bytes_per_pixel(bmp_header)) % 4) % 4;
}
```

```
int _get_bytes_per_pixel(BMPHeader  *bmp_header){
   return bmp_header->bits_per_pixel / BITS_PER_BYTE;
}

int _get_position_on_row(int x, BMPHeader *bmp_header){
   return x * _get_bytes_per_pixel(bmp_header);
}

bool _check(bool condition, char **error, const char *error_message){
   bool is_valid = true;
   if(!condition)    {
      is_valid = false;
      if (*error == NULL)  // to avoid memory leaks
      {
         *error = error_message;
      }
   }
   return is_valid;
}
```

#bitmap.h

```
#ifndef BITMAP_H
#define BITMAP_H

#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>

#define BMP_HEADER_SIZE 54
#define DIB_HEADER_SIZE 40

#pragma pack(push)  // save the original data alignment
#pragma pack(1)     // Set data alignment to 1 byte boundary

typedef struct {
```

```
    uint16_t type;           // Magic identifier: 0x4d42 "BM"
    uint32_t size;           // File size in bytes
    uint16_t reserved1;      // Not used
    uint16_t reserved2;      // Not used
    uint32_t offset;         // Offset to image data in bytes from beginning of file
    uint32_t dib_header_size;   // DIB Header size in bytes
    int32_t  width_px;       // Width of the image
    int32_t  height_px;      // Height of image
    uint16_t num_planes;     // Number of color planes
    uint16_t bits_per_pixel;    // Bits per pixel
    uint32_t compression;    // Compression type
    uint32_t image_size_bytes;  // Image size in bytes
    int32_t  x_resolution_ppm;  // Pixels per meter
    int32_t  y_resolution_ppm;  // Pixels per meter
    uint32_t num_colors;     // Number of colors
    uint32_t important_colors;  // Important colors
} BMPHeader;

#pragma pack(pop)  // restore the previous pack setting

typedef struct {
    BMPHeader header;
    unsigned char* data;
} BMPImage;

BMPImage* read_bmp(FILE* fp, char** error);
bool      check_bmp_header(BMPHeader* bmp_header, FILE* fp);
bool      write_bmp(FILE* fp, BMPImage* image, char** error);
void      free_bmp(BMPImage* image);
BMPImage *rotateRight(BMPImage *image, char **error);

#endif /* BITMAP_H */
```