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"

void read\_image(const char \*filename, BMPImage \*\*image);

void write\_image(const char \*filename, BMPImage \*image);

FILE \*\_open\_file(const char \*filename, const char \*mode);

void \_clean\_up(FILE \*fp, BMPImage \*image, char \*\*error);

int main(void) {

BMPImage \*image;

read\_image("ind.bmp", &image);

Internal \*in = bmp\_to\_Internal(image);

Internal \*out = rotateRight(in);

BMPImage \*n\_image = internal\_to\_BMP(out);

write\_image("out.bmp", n\_image);

return EXIT\_SUCCESS;

}

void read\_image(const char \*filename, BMPImage \*\*image) {

char \*read\_status[] = { "Not enough memory", "Cannot read header",

"Invalid BMP file", "Cannot read image", "Image is read successfully"};

FILE \*input\_ptr = \_open\_file(filename, "rb");

enum READ\_STATUS stt = read\_bmp(input\_ptr, image);

if (stt != READ) {

fprintf(stderr, "ERROR: %s\n", read\_status[stt]);

} else printf( "%s\n", read\_status[stt]);

}

void write\_image(const char \*filename, BMPImage \*image) {

char \*write\_status[] = { "Cannot write image", "Image is written successfully"};

FILE \*output\_ptr = \_open\_file(filename, "wb");

enum WRITE\_STATUS stt = write\_bmp(output\_ptr, image);

if (stt != WRITTEN) {

fprintf(stderr, "ERROR: %s\n", write\_status[stt]);

} else printf( "%s\n", write\_status[stt]);

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 \_clean\_up(FILE \*fp, BMPImage \*image, char \*\*error) {

if (fp != NULL) {

fclose(fp);

}

free\_bmp(image);

}

**#bitmap.c**

#include "bitmap.h"

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

enum READ\_STATUS read\_bmp(FILE \*fp, BMPImage \*\*image)

{

\*image = malloc(sizeof(BMPImage));

if (image == NULL)

{

return MEMORY\_NOT\_ENOUGH;

}

rewind(fp);

if(fread(&((\*image)->header), sizeof((\*image)->header), 1, fp) != 1)

{

return READ\_HEADER\_ERROR;

}

if(!check\_bmp\_header(&((\*image)->header), fp))

{

return INVALID\_BMP\_FILE;

}

(\*image)->data = malloc(((\*image)->header).image\_size\_bytes);

if ((\*image)->data == NULL)

{

return MEMORY\_NOT\_ENOUGH;

}

if (fread((\*image)->data, ((\*image)->header).image\_size\_bytes, 1, fp) != 1)

{

return READ\_IMAGE\_ERROR;

}

return READ;

}

bool write\_bmp(FILE \*fp, BMPImage \*image)

{

// Write header

rewind(fp);

if (fwrite(&image->header, sizeof(image->header), 1, fp) != 1)

{

return CANNOT\_WRITE\_IMAGE;

}

// Write image data

if (fwrite(image->data, image->header.image\_size\_bytes, 1, fp) != 1)

{

return CANNOT\_WRITE\_IMAGE;

}

return WRITTEN;

}

Internal \*bmp\_to\_Internal(BMPImage \*image){

Internal \*out = malloc(sizeof(Internal));

out->data = image->data;

out->height\_px = image->header.height\_px;

out->width\_px = image->header.width\_px;

return out;

}

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 \*internal\_to\_BMP(Internal \*in){

BMPImage \*new\_image = malloc(sizeof(BMPImage));

new\_image->header.type = MAGIC\_VALUE;

new\_image->header.offset = BMP\_HEADER\_SIZE;

new\_image->header.dib\_header\_size = DIB\_HEADER\_SIZE;

new\_image->header.num\_planes = NUM\_PLANE;

new\_image->header.compression = COMPRESSION;

new\_image->header.num\_colors = NUM\_COLORS;

new\_image->header.important\_colors = IMPORTANT\_COLORS;

new\_image->header.bits\_per\_pixel = BITS\_PER\_PIXEL;

new\_image->header.image\_size\_bytes = \_get\_internal\_size\_bytes(in);

new\_image->header.size = new\_image->header.image\_size\_bytes + BMP\_HEADER\_SIZE;

new\_image->header.x\_resolution\_ppm = 0;

new\_image->header.y\_resolution\_ppm = 0;

new\_image->header.height\_px = in->height\_px;

new\_image->header.width\_px = in->width\_px;

new\_image->data = in->data;

return new\_image;

}

Internal \*rotateRight(Internal \*in) {

Internal \*out = malloc(sizeof (Internal));

int y = 0;

int x = in->width\_px;

int w = in->height\_px;

int h = in->width\_px;

out->width\_px = in->height\_px;

out->height\_px = in->width\_px;

out->data = malloc(\_get\_internal\_size\_bytes(in));

int position\_y = y \* \_get\_internal\_row\_size\_bytes(in);

int position\_x\_row = \_get\_position\_on\_row(x)-3;

int new\_index = 0;

for (int i = 0; i < h; i++) {

for (int j = 0; j < w; j++) {

for (int k = 0; k < 3; k++) {

out->data[new\_index] = in->data[position\_y + position\_x\_row];

new\_index++;

position\_x\_row++;

}

position\_y += \_get\_internal\_row\_size\_bytes(in);

position\_x\_row = \_get\_position\_on\_row(x)-3;

}

int padding = \_get\_padding\_internal(out);

for (int l = 0; l < padding; l++) {

out->data[new\_index] = 0x00;

new\_index++;

}

position\_y = y \* \_get\_internal\_row\_size\_bytes(in);

x--;

position\_x\_row = \_get\_position\_on\_row(x)-3;

}

return out;

}

long \_get\_file\_size(FILE \*fp)

{

long current\_position = ftell(fp);

if (current\_position == -1)

{

return -1;

}

if (fseek(fp, 0, SEEK\_END) != 0)

{

return -2;

}

long file\_size = ftell(fp);

if (file\_size == -1)

{

return -3;

}

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\_internal\_size\_bytes(Internal \*in)

{

return \_get\_internal\_row\_size\_bytes(in) \* in->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\_internal\_row\_size\_bytes(Internal \*in)

{

int bytes\_per\_row\_without\_padding = in->width\_px \* \_get\_bytes\_per\_pixel();

return bytes\_per\_row\_without\_padding + \_get\_padding\_internal(in);

}

int \_get\_padding(BMPHeader \*bmp\_header)

{

return (4 - (bmp\_header->width\_px \* \_get\_bytes\_per\_pixel(bmp\_header)) % 4) % 4;

}

int \_get\_padding\_internal(Internal \*in)

{

return (4 - (in->width\_px \* \_get\_bytes\_per\_pixel()) % 4) % 4;

}

int \_get\_bytes\_per\_pixel()

{

return 3;

}

int \_get\_position\_on\_row(int x)

{

return x \* \_get\_bytes\_per\_pixel();

}

**#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

#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

#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;

typedef struct {

unsigned char\* data;

int32\_t width\_px; // Width of the image

int32\_t height\_px;

} Internal;

struct Pixel {

unsigned char b, g, r;

};

enum READ\_STATUS {

MEMORY\_NOT\_ENOUGH, READ\_HEADER\_ERROR, INVALID\_BMP\_FILE, READ\_IMAGE\_ERROR, READ

};

enum WRITE\_STATUS{

CANNOT\_WRITE\_IMAGE, WRITTEN

};

enum READ\_STATUS read\_bmp(FILE \*fp, BMPImage \*\*image);

bool write\_bmp(FILE \*fp, BMPImage \*image);

bool check\_bmp\_header(BMPHeader \*bmp\_header, FILE \*fp);

void free\_bmp(BMPImage \*image);

long \_get\_file\_size(FILE \*fp);

int \_get\_image\_size\_bytes(BMPHeader \*bmp\_header);

int \_get\_internal\_size\_bytes(Internal \*in);

int \_get\_image\_row\_size\_bytes(BMPHeader \*bmp\_header);

int \_get\_internal\_row\_size\_bytes(Internal \*in);

int \_get\_bytes\_per\_pixel();

int \_get\_padding(BMPHeader \*bmp\_header);

int \_get\_padding\_internal(Internal \*in);

int \_get\_position\_on\_row(int x);

Internal \*bmp\_to\_Internal(BMPImage \*image);

BMPImage \*internal\_to\_BMP(Internal \*in);

Internal \*rotateRight(Internal \*in);

#endif /\* BITMAP\_H \*/