French Azerbaijani University
Arash Habibi
Second year
2017-2018

Advanced C Programming

Practical Work 1 : structures and modular programming

1 Colors

We consider that a color is characterized by three integer components : red, green and blue, all three in the [0,255] interval. Any color can be obtained by a specific additive mixture of these three components. For example yellow is a mixture of red and green without any blue. White is a mixture of all three.

1. In a file called `color.c`, define a structure `struct color` characterized by 3 integer fields : `red, green` and `blue`.
2. Write a `main` function in which a variable of type `struct color` is declared. In the rest of this exercice, Each time you have written a function, make sure that you have tested it by different calls in the main function.
3. We wish to write a function with prototype `void C_print(struct color c);` which would output on the terminal the three components of color `c`. For example if all three components of `c` were equal to 255, then `C_print(c)` would print `(255,255,255)`. Compile and test.
4. In the `main` function, we had declared a `struct color` variable (question 2). Print the components of that color by a call to `C_print`.
5. We wish to write a function with prototype `struct color C_new(int r, int g, int b);`. For 3 given integers, this function would return a color variable with those components. In the `main` function, add the following calls :

   ```
   struct color orange = C_new(255,100,0);
   C_print(orange);
   ```

   Make sure your program is well compiled and that the output is coherent.
6. In the `color.c` source file, write a function with prototype `int clamp(int n);`. This function must always return a value between 0 and 255.
   — If `n` is negative, the function must return 0 ;
   — If `n` is greater than 255, the function must return 255 ;
   — In all other cases (normal cases) the function must return `n` unchanged.
   Modify `C_new` so that if any of the three parameters of this function is outside the [0,255] interval, the components of the returned color are placed again in this valid interval. In order to test this function, change the previous call and let the red component of `orange` be 300 (too high).

   ```
   struct color orange = C_new(300,100,0);
   C_print(orange);
   ```

   `C_print` should print `(255,100,0)`. For all following functions, we assume that the components are within [0,255].
7. Write a function with prototype `struct color C_multiply(struct color c, float coef);`. For a given `c` color and a positive coefficient `coef`, this function returns a color whose components are those of `c` multiplied by `coef`. Make sure that even if `coef` is greater than 1, the components of the returned color are still within [0,255]. (Hint : you should not need to make more tests, you have already written the test somewhere. Remember where ?)

8. Write a function with prototype `struct color C_negative(struct color c);` which, for a light color returns a dark color and vice-versa. More specifically, if `r` is the red component of color `c`, then the red component of the returned color should be `255 - r`. Same thing for the other components.
9. Write a function with prototype `struct color C_permute(struct color c);` which permutes the components. If the components of `c` are `r`, `g` and `b`, then the components of the returned color should be `g`, `b` and `r`.
10. Write a function with prototype `int C_intensity(struct color c);`. For a given `c` color this function returns an integer which represents the average of the three color components of `c`. For example for color `C_new(58,58,58)`, this function should return 58. For color `C_new(255,0,255)`, this function should return (255+0+255)/3 = 170.
11. Gray colors range from very light gray, almost white, to dark gray, almost black. Gray is defined by the fact that the red, green and blue components are equal. For example (5,5,5) is a very dark gray, whereas (220,220,220) is almost white. Write a function with prototype `struct color C_grayScale(struct color c);`. For a given `c` color this function returns a gray color whose components are defined by the intensity of `c`.
12. Write a function with prototype `int C_threshold(struct color c, int th);`. For a given `c` color and an integer `th` this function compares the intensity of `c` with `th`.
    — If the intensity of `c` is higher than `th`, the function should return 255 ;
    — Otherwise, it should return 0.

2 Conditional compilation and *include guards*

Consider a header file `foo.h` (left) which only defines the `foo` structure. Next, consider the source file named `foo.c` (right) which only declares a variable of type `struct foo`.

```
struct foo
{ int x,y; };
```

```
#include "foo.h"

int main()
{    struct foo f;
     return 0
}
```

1. Does the compilation generate error messages or warnings ?
2. What if, in the source file, we included the header file twice ?

```
struct foo
{ int x,y; };
```

```
#include "foo.h"
#include "foo.h"

int main()
{    struct foo f;
     return 0
}
```

3. Now we change the header file to the following content. What is the meaning of the conditional compilation directives ?

```
#ifndef BLAH
#define BLAH

struct foo
{ int x,y; };

#endif
```

```
#include "foo.h"
#include "foo.h"

int main()
{    struct foo f;
     return 0
}
```

In many advanced header files, you will often find such directives which are called *include guards* and which make sure that the header file is never included more than once.

3 Colors : modular programming

We have already written a program dealing with colors. Let us now put it in several module files.

1. Open an empty file and call it `color.h`. Put in that file the definition of the color structure and all of the function prototypes. Remove these elements from `color.c`;
2. Include this header file in `color.c`. Compile and test.
3. Open an empty file and call it `main.c`. Remove the main function from `color.c` and put it in `main.c`. Compile and test.
4. We want the clamp function to be static (visible to only the functions in `color.c`). It must be removed from the header file, and it must be declared static. Compile and test your functions.

4 Images

We consider that an image is simply an array of colors.

1. Open an empty header file (name it `image.h`) and write :

   ```
   #include <stdio.h>
   #include "color.h"
   // because image functions will also necessarily deal with colors
   ```

   But before adding any function definitions or prototypes in this module, let us include this small header file in `main.c`, since the `main` function will call functions dealing with colors as well as images. Thus in `main.c` we have included three headers :

   ```
   #include <stdio.h>
   #include "color.h"
   #include "image.h"
   ```

   Compile the program. Where does the problem come from ? Hint : `image.h` contains an include of `color.h`. This means that `color.h` has been included twice in `main.c`. How can we avoid this ? (Think again about what we saw in section 2 about include guards). Find a way to include `color.h` and `image.h` in `main.c`.
2. In `image.h` add the following function declarations :

   ```
   void I_print    (struct color img[], int nb_pixels);
   void I_coef     (struct color img[], int nb_pixels, float coef);
   void I_negative (struct color img[], int nb_pixels);
   void I_permute  (struct color img[], int nb_pixels);
   void I_grayScale(struct color img[], int nb_pixels);
   void I_threshold(struct color img[], int nb_pixels, int th);
   ```

   Compile. Every thing should go right.
3. Open an empty file (name it `image.c`), include `image.h` and define the 6 previous functions. Each of them simply steps through the colors of the array and applies to each color the corresponding color function : `I_print` applies `C_print` to each color, and same thing for `I_coef`, `I_negative` and `I_permute` etc. Compile.
4. In order to test these functions, go to the `main` function and create an array `img` of 8 colors ; This could represent a tiny image of 4×2 pixels.
5. In the same file, initialize that array, for example in the following way :

```
img[0] = C_new(0,0,0);       // black
img[1] = C_new(255,0,0);     // red
img[2] = C_new(0,255,0);     // green
img[3] = C_new(0,0,255);     // blue
img[4] = C_new(255,255,0);   // yellow
img[5] = C_new(255,0,255);   // magenta
img[6] = C_new(0,255,255);   // cyan
img[7] = C_new(255,255,255);// white
```

6. Test your six image functions with this small array.

5 PPM image file format : testing your functions with real images

Now let us apply these image processing functions to real images.

Reading and writing in a ppm image file is beyond the scope of this exercice session. You will use already prepared functions in the `ppm.h` and `ppm.c` files that you can download from the *Moodle* platform. You can also download a ppm image file : `merida.ppm`. All you need to know to use ppm functions is explained in the header file.

1. include `ppm.h` in `main.c` and compile the 4 modules : `color.c`, `image.c`, `ppm.c` and `main.c`.
2. Now we can call `ppm` functions in `main`. The typical use of these functions is :

```
struct color img[];
int nbpixels;
struct ppm p;

p = PPM_new("merida.ppm"); // reads the file and puts everything in p
nbpixels = PPM_nbPixels(p);
img = PPM_pixels(p);         // img is the color array

... // transform the image with your image functions

PPM_save(p,img,"res.ppm"); // saves the result in res.ppm
```

Process the image file with each of the 6 functions written in the previous section. Make sure that your results are similar to those of figure 1.

FIGURE 1 – From left to right : The original image, gray-scale, negative, motion-blur. On the lower row, threshold, permute, composed and gradient.

3. In the `color.c` source file, write a function with prototype

```
int C_areEqual(struct color c1, struct color c2):
```

which returns 1 if each of the three components of `c1` equals the corresponding component of `c2`. For example :

```
struct color c1 = C_new(10,50,200);
struct color c3 = C_new(20,50,200);
struct color c3 = C_new(10,50,200);
int a = C_areEqual(c1,c2);      // a is 0
int b = C_areEqual(c1,c3);      // b is 1
```

Do not forget to add the prototype in the `color.h` header file.

4. In the `image.c` file, write a function with prototype :

```
void I_compose(struct color img1[], struct color img2[], int nb_pixels,
               struct color target);
```

The arguments of this function are two images with the same size (`nb_pixels` pixels) and one `target` color. This function loops through `img1`. Each time it detects pixels whose color is equal to the target color, it replaces that color by the color of the same pixel in the second image `(img2)`. For example, if `img1` is the picture on the upper left corner of figure 1 and if `img2` is the image of a forest (both images have the same size) and `target` is the green color in the background of `img1` (that is `C_new(0,111,92)`), this function replaces the background green by the image of the forest. This produces seventh image of figure 1.

6 Writing convolution filters

Up to this point, we have processed the pixels individually regardless of each pixel's neighbors. But for many image processing operations (called *filters*) the color of each output pixel is not determined by the color of one input pixel, but by the color of several pixels. This means that we must not directly change the values of the input pixel array. We must have an input pixel array and an output pixel array.

Moreovre, the pixels are all in a one-dimensional array. So it is easy to know a pixel's left and right neighbors. It is more difficult to know a pixel's upper and lower neighbors. But we can already make interesting operations by using the left and right neighbors. For example making a motion-blur effect implies to replace a pixel's value with the average of the neighboring pixels. Or detecting contours involves the difference between neighboring pixels.

1. In the `color.c` file, write a function with prototype

```
struct color C_addCoef(struct color c1, struct color c2, float coef);
```

which adds to the components of `c1`, the color produced by `coef` times the components of `c2`. For example for `coef=1`, this function returns the sum of `c1` and `c2`. For `coef=-1` this function returns the substraction `c1-c2`. For `coef=0.5` this function returns `c1 + 0.5 * c2`.

2. In the `image.c` file, write a function with prototype

```
void I_addColor(struct color img[], int nb_pixels, struct color c);
```

which adds to each pixel of `img` the same color `c`. For example `I_addColor(img, C_new(0,0,0));` has no effect on `img`, but `I_addColor(img, C_new(127,127,127)` adds a medium gray color to each pixel of `img`.

3. In the `image.c` file, write a function with prototype

```
void I_gradient(struct color img_in[], struct color img_out[], int nb_pixels);
```

The `img_out` array is supposed to be defined before the call of this function. This function defines the medium gray color :

```
struct color gray = C_new(127,127,127);
```

It puts in each pixel of `img_out` the difference between the corresponding pixel of `img_in` and its left neighbor. And since this may produce negative results we add `gray` to the resulting difference. For the pixels at the left side of the image, we consider that the left neighbor is the pixel at the right side of the image on the previous line. In other words, the left neighbor of `img_in[i]` is always `img_in[i-1]`, except for `i=0`. We assume that `img_out[0]` is equal to `gray`. Test this function on *Merida* (cf figure 1).

4. In the `image.c` file, write a function with prototype

   ```
   void I_gradient(struct color img_in[], struct color img_out[], int nb_pixels);
   ```

   The `img_out` array is supposed to be defined before the call of this function. This function puts in each pixel `img_out[i]` the difference between `img_in[i]` and `img_in[i-1]` (i.e. the difference between the corresponding pixel in `img_in` and its left neighbor). We assume that `img_out[0]` is always black. After this process, many pixels may have negative values. In order to avoid these negative values, the function adds the medium gray `C_new(127,127,127)` to all of the pixels of `img_out`. Test this function on `merida.ppm`.

5. In the `image.c` file, write a function with prototype

   ```
   struct color I_average(struct color img[], int nb_pixels, int fromhere,
                          int nb_pixels_average);
   ```

   which operates on an image `img` containing `nb_pixels` elements. It returns a color which is the average of the colors of the `nb_pixels_average` pixels between pixel number `fromhere` and pixel number `fromhere+nb_pixels_average-1`. For example if `fromhere=76` and `nb_pixels_average=5` then this function returns the average of the pixels number 76, 77, 78, 79, 80 and 81. We assume that

   ```
   fromhere+nb_pixels_average < nb_pixels
   ```

6. In the `image.c` file, write a function with prototype

   ```
   void I_motionBlur(struct color img[], struct color img_out[],
                     int nb_pixels, int strength);
   ```

   which operates on an image `img` containing `nb_pixels` elements. It puts in each pixel `img_out[i]` the average of the `strength` following pixels in `img`. If there are less than `strength` pixels after `img[i]`, then it returns the average of the pixels between `img[i]` and the end of `img`. Test this function on `merida.ppm`.

Congradulations if you have reached this point. You have written a quite elaborate set of image processing tools.