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 admit 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 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`.

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);
   ```

   Compile. Every thing should go right.

3. Open an empty file (name it `image.c`), include `image.h` and define the 4 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`. 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 four image functions with this small array.

5 PPM image file format

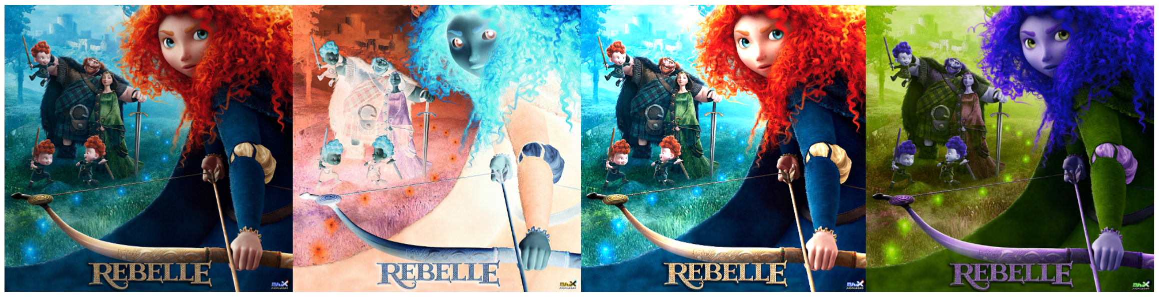

FIGURE 1 – From left to right : The original image, negative, brighten, and permute

Now let us apply these image processing functions to real images. This will also give us an opportunity to build a library and to use it.

Reading and writing in a ppm image file is beyond the scope of this exercice session. All you need has been written in the `ppm.h` and `ppm.c` that you can download from the *Moodle* platform. You can also download a ppm image file : `rebelle.ppm`. Consider first `ppm.h`.

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

#define PGM_MAX 300

struct ppm
{       FILE *ppmfile;
        int width, height, nbpixels;
        struct color *pixels;
};

// Given the path of a ppm image file, this function
// reads this file and stores the information in a ppm structure
struct ppm PPM_new(char *ppmfilepath);

// Given a ppm structure initialized by PPM_new,
// PPM_nbPixels returns the number of pixels in the image
int  PPM_nbPixels(struct ppm p);

// Given a ppm structure initialized by PPM_new,
// PPM_pixels returns an array of colors representing the images
struct color* PPM_pixels(struct ppm p);

// Given a ppm structure initialized by PPM_new,
// and an array of colors image, PPM_save saves this new file
// (with the same width and height) in the file whose path is given
// in the outputfile parameter
void PPM_save(struct ppm p, struct color *image, char *outputfile);
```

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("rebelle.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 and see the results.
3. Now, suppose that you would like other people to be able to use the `ppm` module.
   — The first step is to compile `ppm.c` without any linking : `gcc -c ppm.c` This generates an object file (`ppm.o`).
   — make a library file from the object file : `ar -cvq libppm.a ppm.o`
   — compile : `gcc color.c image.c main.c -L. -lppm`. The `-L.` option specifies that the library is in the current directory. If you could place the library `libppm.a` in a directory as in `/usr/local/lib`, then there would be no need for this compilation option, because the compiler automatically searches this directory.

With this library, you can use PPM functions in other programs without necessarily having to copy ppm.c and ppm.h in your project and without having to compile them either.