Hardware-Software Interface: Coursework 1.

Daniyar Nazarbayev.

H00204990.

Table of contents

1. Code.

```
# include <stdlib.h>
# include <stdio.h>
# include <string.h>
# include <stddef.h>
# include <inttypes.h>
//# include <limits.h>

struct PPM;
struct pixels;

char** splitString(char*, char*);
struct PPM * getPPM(FILE*);
void showPPM(struct PPM*);
struct PPM * copyPPM(struct PPM *);
struct PPM * encode(char*, struct PPM *);
void showPPM(struct PPM*);
char* decode(struct PPM*, struct PPM*);
int difference(struct PPM*, struct PPM*);
void makePPM(struct PPM*);
int duplicate(int[], int, int);
//void releasePPM(struct PPM*);
int contains(char*, int, char*, int);
int getLine(FILE*, char*, int);
int RNJesus(int, int);
```

```
// minimum 3 arguments required
int main(int argc, char** argv){
	//printf("%llu\n", RAND_MAX); // = 32767
	//printf("%"PRId64"\n", RAND_MAX);
	//printf("%d\n", INT_MAX);
	if(argc>3){
		// args[1] is d for decode
		// args[2] is original ppm
		// args[3] is altered ppm
		if(strcmp(argv[1], "d") == 0){
			FILE* ppm1 = fopen(argv[2], "r");
			struct PPM* original = getPPM(ppm1);
			fclose(ppm1);

			FILE* ppm2 = fopen(argv[3], "r");
			struct PPM* altered = getPPM(ppm2);
			fclose(ppm2);

			printf("%s\n", decode(original, altered));
		}
		// args[1] is e for encode
		// args[2] is original ppm
		// args[3] is text you want to encode
		else if(strcmp(argv[1], "e") == 0){
			FILE* ppm1 = fopen(argv[2], "r");
			struct PPM* original = getPPM(ppm1);
			fclose(ppm1);
```

```
                        struct PPM* altered = encode(argv[3], original);
                        makePPM(altered);
                }
        }
        return 0;
}

struct pixel{
        unsigned char red;
        unsigned char green;
        unsigned char blue;
};

struct PPM {
        char* ext_code;
        char** comments;
        int number_of_comments;
        int width;
        int height;
        int max;
        struct pixel** pixels; // i wanted to do pixels[width*height], but it does not compile, so i will allocate memory after i create an instance of a struct.
};

struct PPM * getPPM(FILE * fd){
        struct PPM* image = malloc(sizeof(struct PPM));

        int n = 50;
```

```
	char temp[n];
	int length = 0;

	length = getLine(fd, &temp[0], n-1);
	if(contains("P3", 2, temp, n)){
		image->ext_code = malloc(length);
		strcpy(image->ext_code, temp);

		// &temp[0] instead of &temp is too avoid a compiler warning
		length = getLine(fd, &temp[0], n-1);
		int i = 0; //comment_counter

		// comments are optional, so i need to check whether they exist first, and only then allocate.
		if(contains("#", 1, temp, n)){
			image->comments = malloc((i+1)*sizeof(char*));
			image->comments[i] = malloc(length);
			strcpy(image->comments[i], temp);
			i++;
			length = getLine(fd, &temp[0], n-1);
			while(contains("#", 1, temp, n)){
				// proper realloc
				char** comments_temp = realloc(image->comments, ((i+1)*sizeof(char*)));
				if(comments_temp!=NULL){
					image->comments = comments_temp;
				} else{
					free(comments_temp);
```

```
                exit(0);
            }
            image->comments[i] = malloc(length);
            strcpy(image->comments[i], temp);
            i++;
            length = getLine(fd, &temp[0], n-1);
        }
    }
    image->number_of_comments = i;
    char** str = splitString(temp, " ");
    image->width = atoi(str[0]);
    image->height = atoi(str[1]);

    length = getLine(fd, &temp[0], n-1);
    image->max = atoi(temp); // atoi to convert to INT

    image->pixels = malloc(image->width * image->height * sizeof(struct pixel*));
    int counter = 0;
    // so that i won't repeat the calculation
    int resolution = image->width * image->height;

    while((length = getLine(fd, &temp[0], n-1)) && counter<resolution){
        image->pixels[counter] = malloc(sizeof(struct pixel));
        if(length>1){
            str = splitString(temp, " ");
            image->pixels[counter]->red = atoi(str[0]);
            image->pixels[counter]->green = atoi(str[1]);
            image->pixels[counter]->blue = atoi(str[2]);
```

```
                        } else{
                                image->pixels[counter]->red = 0;
                                image->pixels[counter]->green = 0;
                                image->pixels[counter]->blue = 0;
                        }
                counter++;
                }
                return image;
        } else{
                exit(0);
        }

}

void showPPM(struct PPM * i){
        printf("%s\n", i->ext_code);

        int counter = 0;
        while(counter < i->number_of_comments){
                printf("%s\n", i->comments[counter]);
                counter++;
        }
        printf("%d %d\n", i->width, i->height);

        printf("%d\n", i->max);
        counter = 0;
        int pixel_counter = i->width * i->height; // so that i would not repeat the calculation
        while(counter<pixel_counter){
```

```
                printf("%d. %d %d %d\n", counter, i->pixels[counter]->red, i->pixels[counter]->green, i->pixels[counter]->blue);
                counter++;
        }
}

/*
// function that i wanted to add to free up the heap.
// not working. program crashes.
void releasePPM(struct PPM* ppm){
        int x = 0;
        while(x<ppm->number_of_comments){
                free(ppm->comments[x]);
                x++;
        }
        free(ppm->comments);
        free(ppm->ext_code);
        x = 0;
        int until = ppm->height*ppm->width;
        while(x<until){
                free(ppm->pixels[x]);
                x++;
        }
        free(ppm->pixels);
        free(ppm);
}

*/
```

```
// deep copy
struct PPM * copyPPM(struct PPM* original){
	struct PPM* copy = malloc(sizeof(struct PPM));
	copy->ext_code = malloc(strlen(original->ext_code)+1);
	copy->ext_code = original->ext_code;
	copy->number_of_comments = original->number_of_comments;

	copy->comments = malloc (original->number_of_comments*sizeof(char*));
	int i=0;
	while(i<copy->number_of_comments){
		copy->comments[i] = malloc(strlen(original->comments[i])+1);
		copy->comments[i] = original->comments[i];
		i++;
	}

	copy->max = original->max;
	copy->width = original->width;
	copy->height = original->height;

	i=0;
	int resolution = copy->width * copy->height;
	copy->pixels = malloc(copy->width * copy->height * sizeof(struct pixel*));
	while(i<resolution){
		copy->pixels[i] = malloc (sizeof(struct pixel));
		copy->pixels[i]->red = original->pixels[i]->red;
		copy->pixels[i]->green = original->pixels[i]->green;
		copy->pixels[i]->blue = original->pixels[i]->blue;
```

```
                i++;
        }
        return copy;
}

struct PPM * encode(char * text, struct PPM * i){
        struct PPM * altered = copyPPM(i);
        int number;
        srand(1);

        // check if there are enough pixels to hold the text
        if(strlen(text) < altered->width * altered->height){
                int x = 0;
                int* rand_numbers = malloc(sizeof(int*));
                rand_numbers[0] = altered->width * altered->height; // using this value because the
random number generater won't be able to get it
                while(x < strlen(text)){
                        number = RNJesus(0, altered->width*altered->height);
                        if((altered->pixels[number]->red != text[x]) && (duplicate(rand_numbers, x+1,
number)==0)){
                                altered->pixels[number]->red = text[x];
                                rand_numbers[x] = number;
                                x++;
                                int* temp = realloc(rand_numbers, ((x+1)*sizeof(int*)));
                                if(temp!=NULL){
                                        rand_numbers = temp;
                                } else{
                                        free(temp);
```

```
                                        exit(0);
                                    }
                                }
                            }
                        }
                        return altered;
}

int duplicate(int* array, int length, int number){
                        int x = 0;
                        while(x<length){
                                    if(array[x]==number){
                                                return 1;
                                    }
                        x++;
                        }
                        return 0;
}

// i1 is original, i2 is altered
char * decode(struct PPM * original, struct PPM * altered){
                        srand(1);
                        int text_length = difference(original, altered);
                        char* text =  malloc(text_length+1);
                        int random;
                        int x = 0;

                        int* rand_numbers = malloc(sizeof(int*));
```

```
	rand_numbers[0] = altered->width * altered->height; // using this value because the random number generater won't be able to get it
	while(x < text_length){
		random = RNJesus(0, original->width * original->height);
		if((original->pixels[random]->red != altered->pixels[random]->red) && (duplicate(rand_numbers, x+1, random)==0)){
			text[x] = altered->pixels[random]->red;
			rand_numbers[x] = random;
			x++;
			int* temp = realloc(rand_numbers, ((x+1)*sizeof(int*)));
			if(temp!=NULL){
				rand_numbers = temp;
			} else{
				free(temp);
				exit(0);
			}
		}
	}
	text[x++] = '\0';
	return text;
}

int difference(struct PPM * i1, struct PPM * i2){
	int x = 0;
	int counter = 0;
	int pix_count = i1->width*i1->height;
	while(x < pix_count){
		if(i1->pixels[x]->red != i2->pixels[x]->red){
```

```
                counter++;
            }
            x++;
        }
        return counter;
}

// for windows - C:/Users/Daniel/Desktop/out.ppm
// for linux - /home/daniel-potter/Desktop/out.ppm ; pwd was used to get that
void makePPM(struct PPM * ppm){
        FILE * output = fopen("/home/daniel-potter/Desktop/out.ppm", "w");
        fprintf(output, "%s\n", ppm->ext_code);
        int x = 0;
        while(x < ppm->number_of_comments){
            fprintf(output, "%s\n", ppm->comments[x]);
            x++;
        }
        fprintf(output, "%d %d\n", ppm->width, ppm->height);
        fprintf(output, "%d\n", ppm->max);
        x = 0;
        int total = ppm->width * ppm->height;
        while(x < total){
            fprintf(output, "%d %d %d\n", ppm->pixels[x]->red, ppm->pixels[x]->green, ppm-
>pixels[x]->blue);
            x++;
        }
        fclose(output);
}
```

```
int RNJesus(int lower_limit, int higher_limit){
	higher_limit -= 1;
	return lower_limit + rand() / (RAND_MAX / (higher_limit-lower_limit+1)+1);
}

char** splitString(char* string, char* delim){
	char** output = malloc(sizeof(char*));
	char* temp;
	int i = 0;
	temp = strtok(string, delim);
	while(temp!= NULL){
		output[i] = malloc(strlen(temp)+1);
		strcpy(output[i], temp);
		temp = strtok(NULL, delim);
		i++;

		char** reallocation = realloc(output, (i+1)*sizeof(char*));
		if(reallocation==NULL){
			free(reallocation);
			exit(0);
		} else{
			output = reallocation;
		}
	}
	return output;
}
```

```
// input for getline's n parameter should be (length of the string - 1)
// this is to null terminate a char array
int getLine(FILE * fin, char a[], int n){
	int counter = 0;
	char ch;
	while((ch = fgetc(fin))!='\n'){
		// use single quotes (') for chars
		if(counter<n){
			a[counter] = ch;
			counter++;
		} else if(ch==EOF){
			a[counter++] = '\0';
			return EOF;
		}
	}
	a[counter++] = '\0';
	return counter;
}

int contains(char target[], int m, char source[], int n){
	int x = 0;
	int y = 0;

	int rollback = 0;

	while(n>x){
		if(source[x]==target[y]){
			rollback = 1;
```

```
                    if((m-1)==y){
                        return 1;
                    }
                    y++;
                } else if (rollback && source[x]!=target[y]){
                    y = 0;
                    rollback = 0;
                    continue;
                }
                x++;
            }
            return 0;
}
```

2. Explanation.

2.1 Note.

First of all, I would like to point out that I used functions getLine() and contains() that I made for lab 2 and splitString(), which is parser, that I made for lab 3.

The PPM file must have 3 RGB values, and data should be separated by new lines, else my parser would be unable to properly parse the file.

The PPM should be big enough in size, meaning that width by height should be greater than the length of the text.

Also, please alter the output path in makePPM() and use sudo when running the executable.

2.2 Pseudo Random.

At first I wondered why do I need the original image to extract the text. Even if I find all the pixels that were changed, I would still have to piece back the data, which at the time I thought was impossible since the pixels were chosen at random. Then I found out that the random number generator in C is actually pseudo random, and gives out the same values depending on what seed it uses.

Seed is the key.

2.3 Length of text.

That made it much more sensible, but I still couldn't figure out the use of the original image, since one could just use the RNG and find all the pixels that were visited. But the more I thought about it, the more it made sense. The original image was there to tell the length of the string, meaning it should tell the random function when to stop.

But, there were still things that did not make sense. What if the letter you want to assign has the same RGB value representation. Personally, I tackled this problem by just skipping that pixel, and calling another rand().

As for the method to decode, I would just count all the pixels (by comparing 2 images), and then use rand() in a loop. The loop has a conditional that checks whether pixels from image 1 and 2 are identical or not, and if they are not then they are one of the characters of the message. Rand() simply puts them in order.

2.4 Least significant bits.

At first, I wanted to implement steganography by changing the last few least significant bits. I figured, since we are doing steganography, then I might as well implement it that way, so that it would not be visible. I thought that changing all 8 bits in RGB (since 1 character is 8 bits) is going to stand out a lot.

That said, the result was surprisingly positive. I guess it is because majority of characters that are used only require 7 bits to represent (ASCII 7):

- 0-9 = 48-57
- A-Z = 65-90
- a-z = 97-122

I want to present one paper about steganography, which I read to understand the coursework better.

Steganography using Cryptography and Pseudo Random Numbers, *by Unik Lokhande and A. K. Gulve*.

In their experiment, they would replace one bit at a random pixel's blue value, and the pixel would either be 4th, 5th or 6th least significant bit. I want to point out some statements that they made:

> "even if the changes are made in LSB's of all blue pixels the change will be very little".
>
> "by using only a single RBG component from the pixel the distortion produced by steganography is negligible".

Considering that we were told to only change the red values of the pixel and majority of the letters in the message will only need 7 bits (meaning that the most significant bit will be 0), meaning that my program, in its current state, can be used in a real life scenario.

In the end, I decided to stick with what the coursework desired. To be honest, I am not too keen to bitwise operators.

2.5 Limiting the rand() range.

At first I wanted to use a 2D array for a pixel, but that overcomplicates things, since rand() gives out only a single number. In the end, I decided to just make a 1D array. All I had to do after that was to somehow limit the range of my random function.

Looking for answers for that was hard. There were no straight forward answers. I knew about the modulo, but I could not use that since I cannot have repetition. I could have just made an if statement, that would check whether the number is greater than width by height, but imagine that it would loop for some time till it gets the value.

In the end, I found it here:

https://stackoverflow.com/questions/1202687/how-do-i-get-a-specific-range-of-numbers-from-rand

One of the answers in that stack overflow question says that the solution to this can be found in the official c-faq:

```
M + rand() / (RAND_MAX / (N - M + 1) + 1)
```

In all honesty, I do not know how it managed to cap the range, but it gets the job done.

The range is inclusive by the way, and that actually caused my program to crash (array out of bounds).

2.6 RAND_MAX.

RAND_MAX is the maximum value rand() can give. Surprisingly on windows it only outputs 32 thousand, which is about 16 bit signed. While on linux, it shows 2 billion, which is 32 bit signed. I researched a bit on that and people say it has something to do with the gcc using old lib or new lib (new lib being the one that has the 32 bit representation).

2.7 Repetition of rand().

At first when googling about the repetition of rand(), the answers that came up were saying that the chances are one in a million.

That said, upon testing my program on very small ppm files, I started getting them. After all, the random function is supposed to narrow down the original rand() input in that range, and if the range is really small, then duplicates are inevitable.

My solution was to create an int array the size of the message. I was not sure whether C allowed an int to point to null, so I decided to use malloc and set the size to the size of a single int pointer, and reallocate it each time a letter is assigned to the struct (each time that happens, a value is incremented, which determines the amount of int pointers).

2.8 Allocating memory for a structure.

Basically, I had a problem, where my code would run in the debugger mode, but would crash in release version mode. It was very hard to determine the cause of the problem because of that. The cause of it was actually that the structure only had 4 bytes allocated to it.

```
struct PPM* ppm = malloc(sizeof(struct PPM*));
```

At the time, I was making a 2D array for comments (array of strings), and this is how I allocated the memory. First for the pointers, and then for each individual pointer. I allocated a struct in a similar manner, but it appeared to be that structs are allocated differently.

The comments - /* and */ helped out a lot to point out the problem. They would not lead you exactly to the line where the problem is, but they will narrow down your search to a single function at least.

I actually found the problem when I was trying to print out height and width, and while width was printing, height would not (and would crash my program). After that I allocated 4 bytes of memory to it … and it still crushed, but when I allocated 5 bytes, it started working.

Another thing I implemented was copyPPM() function, which should make a deep copy. It's not essential to the program , but it was a good learning experience.

I also wanted to implement a releasePPM(), which should release the memory the structs are taking up. The function is not working and is making my program crash currently, but I decided to leave it there nevertheless (it is commented out).

2.9 Final thoughts: compression, stego-key.

Anyway, everything else is straight forward. PPM is a simple format that stores every RGB value. I actually thought that it has no compression. It shows every RGB value (meaning all 24 bits), which is why I thought so. That said, the size of the input file and the output file differ.

The data is separated by new lines, and we had to make a reader that would read until a new line for lab 2. We also had to make a contains function, that should show strings that contain certain characters. That was very useful to get comments actually. In my lab 3, I made a splitString function, which should take a string and cut it into multiple strings. I actually used strtok to achieve that, but with my current skills I think I can create my own.

There was also another thought that had, which was about stego-keys. What would one use as a key that would be known by both parties. At first I thought about the data stored in the PPM itself, but most of it is accessible to everyone, meaning that whoever would try to find the message would first try to input those values (height, width, RGB, max) as seeds, before brute forcing. That said, I suppose one could use comments, with some cryptography applied to them. That said, it is not part of our coursework, and in all honesty, I feel tired after writing 400 lines of code, so maybe another time.