# Lab 4 – Bug Fixing
CPSC 1021- Fall 2019

## Introduction

In today's lab, we will work on debugging and fixing syntax errors in C code. We will also talk about memory leaks and segmentation faults.

## Due: Sunday, September 29, 2019

## Lab Objectives

- Manipulate C style pointers
- Read in and Write out to files
- Use standard I/O functions (fscanf, fprintf, etc)
- Use GDB and Valgrind
- Work with a non-trivial program to identify bugs and read error messages

## Lab Instructions

You will be examining a simple program that can **encrypt** and **decrypt** images files. You will be provided with the following files:

**lab4.c**
**crypto.c**
**crypto.h**
**secret.ppm**
**key**

You will be **fixing compile and run-time errors** in the files **crypto.c** and **lab4.c.** There are no errors in the header file.

To compile the program, use the **gcc** command:

```
gcc -Wall -g lab4.c crypto.c -o crypto
```

Your first goal should be to fix any compile time errors present in the program. This involves reading through the code and any compiler error messages to find errors. Read through the source files and see if you can understand how the program works. A brief explanation follows:

## Image Cryptography

Cryptography in the context of images is the process of transforming (encrypting) a given image such that it is meaningless without being decrypted. On the SoC servers, use the command **display** to view the image **lab4.ppm** you downloaded:

```
display lab4.ppm
```

PPM images (Portable Pixel Map) are images which have basically no encoding or compression. A PPM file consists of 2 parts: 1) A **header** which gives you descriptive information about the image such as dimensions and color profile 2) A data section with raw pixel data. An image is made up of thousands of pixels, with each pixel having a red, green, and blue value. It is not necessary to completely understand PPM images for this lab, but knowing the basic components will help understand the program. For additional info, follow the link below:

http://netpbm.sourceforge.net/doc/ppm.html

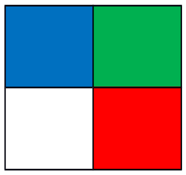
The lab4 program takes a PPM image and will either **encrypt** or **decrypt** it when supplied with a **key**.

A **key** in cryptography is much like its real-world counterpart; to unlock/decrypt a given image, you must have the key that was used to encrypt it in the first place. This type of cryptography is known as **symmetrical key cryptography**, because both encrypter/decrypter must have access to the same key.

The goal of this lab is to fix the program and decrypt the secret PPM image. **This image will contain a number which you should submit in a README file along with the completed source code.**

## The Algorithm

The program works by randomly swapping pixels in the source image, using the **location** of the pixel and the **key** to allow for the swap operations to be reconstructed. For example, imagine we had an image with 4 pixels that looks like this:



The PPM file for this image would look like this:

```
P3 2 2 255
0  0  255
0  255   0
0  0  0
255  0  0
```

Notice the header line and then 4 lines of pixel data. The encryption works like this:

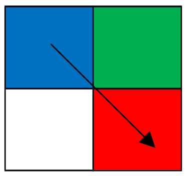
```
for every row of the image from 0 -> # rows
    for every column of the image 0 -> # columns
        generate pseudo-random row coordinate using current row and seed
        generate pseudo-random column coordinate using current column and seed
        swap current pixel with randomly selected pixel
```

Therefore, this 4-pixel image will experience 4 swaps. The decryption works like this:

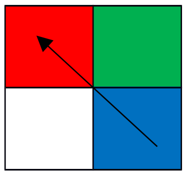
```
for every row of the image from # rows -> 0
    for every column of the image # columns -> 0
        generate pseudo-random row coordinate using current row and seed
        generate pseudo-random column coordinate using current column and seed
        swap current pixel with randomly selected pixel
```

Notice that these two operations are almost the exact same. In encryption, we process pixels from top-left to bottoms-right, while decryption goes in reverse from bottom-right to top-left.

To see why this works, notice that we are generating **pseudo-random** numbers. That is, they are not truly random, since they're value is exactly determined by the **key** value as well as the coordinates of the pixel in question. This allows us to **reverse** all the swaps we perform. For example, say the **bottom-right** pixel randomly generated a swap with the top-left pixel:



**swap bottom-right and top-left**



Decryption just reverses the swaps!

When we decrypt, notice that we work backwards: Taking the **bottom-right** pixel again, we will "randomly" generate the **same swap operation** as before. This is because the key, row, and column are the same. It is easy to see then that working backwards will swap all pixels back to their original orientation.

It's okay if you don't completely understand this algorithm (or maybe don't believe me ◀◀). You just need some content before looking at the code. Part of this assignment is coming into some source code you've never seen and making sense out of it!

## Running the Program

Once you've fixed compiler errors, you can run the program like this:

```
./crypto secret.ppm key decrypted.ppm
```

This will use **secret.ppm** as the source, **key** as the crypto key, and **decrypted.ppm** as the location of the generated image. You will be presented with an option to **encrypt** or **decrypt.** **Do not encrypt an already encrypted image!**

At this stage, you will run into segmentation faults. The program will also have memory leaks. Fix these and then attempt to decrypt the secret message!

## Valgrind and GDB

To identify the cause of segmentation faults, you can use GDB or Valgrind. To use GDB, simply enter the command in your terminal:

```
gdb ./crypto
```

Once you have entered the GDB prompt, use the run command, providing any command line arguments you need. For example:

```
gdb > run secret.ppm key decrypted.ppm
```

This will show you where the segmentation fault occurred. There are many other commands and tools GDB provides. For more info, check the following resources.

http://www.cprogramming.com/gdb.html

Valgrind also allows us to **check for memory leaks.** Remember that a memory leak is when you **malloc** memory but never **free** it. Make sure you identify any memory leaks by running the valgrind command like so:

```
valgrind –leak-check=yes ./crypto secret.ppm key decrypted.ppm
```

Your program should be leak-free and not seg-fault.

## Your Objectives

- Fix all compiler errors and warnings
- Read source code
- Fix all segmentation faults and plug memory leaks using GDB and Valgrind
- Run the decryption program to decrypt secrets.ppm
- Submit the **corrected source and header files** along with a **README.txt** that contains the secret number (lab4.c crypto.c crypto.h README.txt)

**SOME HINTS!**
The source files have comments which show what sections are **free of errors**. You do not have to consider these sections of code other than to understand how the program works.

There will only be errors in:

- lab4.c – main() method
- crypto.c – read_image() method
- crypto.c – sym_crypt() method

These errors will generally be syntactical, but some are bugs which require understanding of how the program works.

## Compile and Execute

Use GCC to compile your code as follows:

```
gcc -Wall -g lab4.c crypto.c -o crypto
```

Execute the program

*./crypto <source ppm> <key file> <output ppm>*

FORMATTING:

1. Your program should be well documented
2. Each file should include a header:
3. Your program should consist of proper and consistent indention
4. No lines of code should be more than 80 characters

```
// Sample Header
/*******************
 your name
 username
 Lab 4
 Lab Section:
 Name of TA
*******************/
```

5 – 10 points will be deducted for each of the above formatting infractions.

Submission Instructions

- Test your program on the School of Computing server prior to submitting.
- Use the tar utility to tar.gz all source files. **Do not tar an entire directory! When I untar your archive, I should see all the files you included, not a top-level directory! Failure to correctly tar will result in a reduction of points!**

  ***EX. tar –czvf username-lab4.tar.gz lab4.c crypto.c crypto.h README.txt***

- Name your tarred file **<username>-lab<#>.tar.gz** (ex. yfeaste-lab4.tar.gz)
- Use handin (http://handin.cs.clemson.edu) to submit your archive