Lab 6.0

Functions, Passing By Reference, and Error Handling

Dr. Chris Bourke

Prior to Lab

Before attending this lab:

1. Read and familiarize yourself with this handout.
2. Read Chapters 5–6 and 18–19 of the Computer Science I textbook
3. Watch Videos 6.1 thru 6.5 of the Computer Science I video series

Peer Programming Pair-Up

To encourage collaboration and a team environment, labs will be structured in a *pair programming* setup. At the start of each lab, you will be randomly paired up with another student (conflicts such as absences will be dealt with by the lab instructor). One of you will be designated the *driver* and the other the *navigator*.

The navigator will be responsible for reading the instructions and telling the driver what to do next. The driver will be in charge of the keyboard and workstation. Both driver and navigator are responsible for suggesting fixes and solutions together. Neither the navigator nor the driver is "in charge." Beyond your immediate pairing, you are encouraged to help and interact and with other pairs in the lab.

Each week you should alternate: if you were a driver last week, be a navigator next, etc. Resolve any issues (you were both drivers last week) within your pair. Ask the lab instructor to resolve issues only when you cannot come to a consensus.

Because of the peer programming setup of labs, it is absolutely essential that you complete any pre-lab activities and familiarize yourself with the handouts prior to coming to lab. Failure to do so will negatively impact your ability to collaborate and work with others which may mean that you will not be able to complete the lab.

1 Lab Objectives & Topics

At the end of this lab you should be familiar with the following

- Basics of enumerated types
- Functions with pass-by-value and pass-by-reference parameters
- How to design and use functions with error handling
- Have exposure to a formal unit testing framework

2 Background

Enumerated Types

Enumerated types are data types that define a set of named values. Enumerated types are often ordered and internally associated with integers (starting with 0 by default and incremented by one in the order of the list). The purpose of enumerated types is to organize certain types and enforce specific values. Without enumerated types, integer values must be used and the convention of assigning certain integer values to certain types in a logical collection must be remembered or documentation referred to as needed. Enumerated types provide a human-readable "tag" to these types of elements, relieving the programmer of continually having to refer to the convention and avoiding errors.

Passing by Value versus Passing by Reference

In general there are two ways in which arguments can be passed to a function: by value or by reference. When an argument (a variable) is passed by value to a function, a copy of the value stored in the variable is placed on the program stack and the function uses this copy's value during its execution. Any changes to the copy are not realized in the calling function, but are local to the function.

In contrast, when an argument (variable) is passed by reference to a function, the variable's memory address is used. Locally (that is, *inside the function*), the variable is treated as a pointer; to get or set its value, it must be dereferenced using the `*` operator. Any changes to the memory cell pointed to by that pointer are reflected globally in the program and more specifically, in the calling function.

A full example:

```
1  //a regular variable:
2  int a = 10;
3
```

```
4  //a pointer to an integer, initially pointing to NULL
5  int *ptr = NULL;
6
7  // & is the referencing operator, it gets the memory
8  // address of the variable a so that ptr can point to it
9  ptr = &a;
10
11 // The star: * is the dereferencing operator.  It changes
12 // the pointer back into a regular variable so a value can
13 // accessed or assigned:
14 *ptr = 20; //a now holds the value 20
```

One of the main uses of pointers is to enable pass-by-reference for functions. Consider the following function:

```
1  int foo(int a, int b) {
2    a = 10;
3    return a + b;
4  }
```

When we call `foo`, the variable's we pass to it are passed *by value* meaning that the value stored in the variables at the time that we call the function are copied and given to the function. The change to the variable `a` in line 2 *only affects the local variable.* The original variable is unaltered. That is, the following code:

```
1  int x = 5;
2  int y = 20;
3  int z = foo(x, y);
4  printf("x, y, z = %d, %d, %d\n" x, y, z);
```

would print `5, 20, 30`: the variable `x` is not changed by the function `foo`. Contrast this with the following:

```
1  int bar(int *a, int b) {
2    *a = 10;
3    return (*a) + b;
4  }
```

In the function `bar`, the variable `a` is passed *by reference* using a pointer. Since the memory address is passed to `bar` instead of a copy of a value, we can alter the original value. Now when line 2 changes the value of the pointer `a`, the change affects the original variable. That is, the code

```
int x = 5;
int y = 20;
int z = foo(&x, y);
printf("x, y, z = %d, %d, %d\n" x, y, z);
```

Will now print `10, 20, 30` since the variable `x` was passed by reference.

Passing by reference also allows us to design functions that can compute multiple values and "return" them in pass-by-reference pointer variables. This frees up the return value to be used as an error code that can be communicated back to the calling function in the event of an error or bad input values.

3 Activities

Clone the GitHub project for this lab using the following URL: https://github.com/cbourke/CSCE155-C-Lab06

3.1 (Re)designing Your Functions

In the previous lab you designed several functions to convert RGB values to gray-scale (using one of three techniques) and to sepia. The details of how to do this available in the previous lab and are repeated for your convenience below.

The design of those functions was less than ideal. For the gray scale functions, there was a function for each of the three techniques. For the sepia conversion, we had to have three separate functions (one for each component, red, green, blue). This was necessary because functions can only return one value. We can't have a function compute and "return" all three RGB values. However, if we use pass-by-reference variables, we *can* compute and "return" multiple values with a single function! Moreover, we can then use the actual return value to indicate an error with the inputs (if any).

Redesign your functions from the previous lab as follows. For the gray scale functionality, implement a single function with the following signature:

```
int toGrayScale(int *r, int *g, int *b, Mode m)
```

The function takes three integer values by reference and will use the value stored in them to compute a gray scale RGB value. It will then store the result *back* into the variables.

To specify which of the three *modes* is to be used, we have defined an enumerated type in the `colorUtils.h` header file:

```
typedef enum {
  AVERAGE,
  LIGHTNESS,
  LUMINOSITY
} Mode;
```

Finally, identify any and all error conditions and use the return value to indicate an error code (0 for no error, non-zero value(s) for error conditions). You should define another enumerated type to represent error codes.

For the sepia filter, implement a single function with the following signature:

```
int toSepia(int *r, int *g, int *b);
```

The function should use the values stored in the variables passed by reference and then store the results in them. Again, error codes should be returned for invalid input values and you should use the enumerated type you defined for error codes.

Finally, add proper documentation to your functions' prototypes.

3.2 Running Unit Tests

As before, you can test your functions using the full image driver program. To build the project use `make` and run the executable `imageDriver`, testing it on a few images of your choice. This is essentially an *ad-hoc test* which is not very rigorous nor reliable and is a manual process.

In the last lab you wrote several *informal* unit tests. Writing unit tests automates the testing process and is far more rigorous. However, this involved writing a lot of boilerplate code to run the tests, print out the results and keep track of the number passed/failed.

In practice, it is better to use a more formal unit testing framework or library. There are several such libraries for C, but one that we'll use is cmocka (https://cmocka.org/). We have provided unit testing code in a file, `colorUtilsTesterCmocka.c` that implements and runs a *suite* of unit tests. You can build this testing suite using :

```
make colorUtilsTesterCmocka
```

and run the resulting executable:

```
./colorUtilsTesterCmocka
```

This starter file should be sufficient to demonstrate how to use cmocka, but the full documentation can be found here: https://api.cmocka.org/. Note that cmocka is already installed on the CSE server. If you compile on your own machine, you will have to install and troubleshoot cmocka yourself.

Run the test suite and verify that your code passes all the tests. Fix any issues or bugs that become apparent as a result of this testing. Show your 100% completed program to a lab instructor and get your worksheet signed off.

Time permitting, add a few of your own test cases to the test suite.

Color Formulas

To convert an RGB value to gray-scale you can use one of several different techniques. Each technique "removes" the color value by setting all three RGB values to the same value but each technique does so in a slightly different way.

The first technique is to simply take the average of all three values:

$$\frac{r+g+b}{3}$$

The second technique, known as the "lightness" technique averages the most prominent and least prominent colors:

$$\frac{\max\{r,g,b\}+\min\{r,g,b\}}{2}$$

The luminosity technique uses a weighted average to account for a human perceptual preference toward green, setting all three values to:

$$0.21r + 0.72g + 0.07b$$

In all three cases, the integer values should be *rounded* rather than truncated.

A sepia filter sets different values to each of the three RGB components using the following formulas. Given a current (r, g, b) value, the sepia tone RGB value, (r', g', b') would be:

$$\begin{aligned} r' &= 0.393r + 0.769g + 0.189b \\ g' &= 0.349r + 0.686g + 0.168b \\ b' &= 0.272r + 0.534g + 0.131b \end{aligned}$$

As with the gray-scale techniques, values should be rounded. If any of the resulting RGB values exceeds 255, they should be reset to the maximum, 255.

4 Advanced Activity (Optional)

Large projects require even more abstraction and tools to manage source code and specify how it gets built. One tool to facilitate this is a program called Make which uses

makefiles–specifications for which pieces of code depend on other pieces and how all pieces get built and combined to build an executable program (or collection of executables). The use of modern IDEs has made the build process much more manageable, but make is still widely used. Several useful tutorials exist describing how makefiles can be defined and used.

- http://www.opussoftware.com/tutorial/TutMakefile.htm
- http://mrbook.org/tutorials/make/

We have provided a makefile for this lab. Read through the tutorials above and then read through the makefile and understand how it works. Write your own makefile for your next assignment to simplify the compilation process.