CSCE 155 - C

Lab 5.0 - Functions

Dr. Chris Bourke

Prior to Lab

Before attending this lab:

1. Read and familiarize yourself with this handout.
2. Read Chapters 5 and 18 of the Computer Science I textbook
3. Watch Videos 5.1 thru 5.6 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:

- Understand how to design, write, and use functions
- Know the difference between a function prototype and function definition
- Modularity and using separate header and source files
- Design test cases and write informal unit tests

2 Background

Most programming languages allow you to define and use functions (or methods). Functions *encapsulate* functionality into a unit of code that can be reused. A function can be specified to take any number of inputs (called parameters or arguments) and return an output. Defining and using functions has several advantages. First, it facilitates code reuse. Rather than cutting and pasting a block of code, it can be encapsulated into a function and reused by calling the function anytime it needs to be executed.

Second, functions facilitate *procedural abstraction*. Often, we don't care or need to worry about the implementation details of a certain algorithm or procedure. By encapsulating the details in a function, we only need to know how to use it (what inputs to provide it and what output we can expect from it). We don't need to worry about *how* it computes its result. For example, up to now you've been using the standard math library's function to compute the square root of a number x, but you haven't had to worry about the details of how this computation actually takes place.

Finally, functions naturally define a *unit* of code that can be easily tested. Typically, unit tests are designed with *test cases* which are input/output combinations that are known to be correct. Unit testing involves feeding the input to a function and comparing the output to the *expected* output. If they match, we say the test case *passes*, if not, we say it *fails*. Unit testing gives a higher level of confidence that our function's implementation is correct.

When defining a function, it is necessary to declare its *signature*. The signature of a function includes:

- The function's identifier – its name
- The return type – the type of variable the function returns
- The parameter list – the number of parameters the function takes (also called its *arity*) along with their types

2.1 Functions in C

In C, functions must be declared before they can be defined or used. Functions are declared by defining *prototypes*: a declaration of the function's signature followed by a semicolon. Prototypes are usually placed into separate header files (with file names ending in `.h`) and their definitions are placed into source files with the same base name (but ending in `.c`). Function definitions repeat the signature but are followed by a block of code that specifies the instructions that will be executed when the function is called or *invoked*.

3 Activities

We have provided partially completed programs for each of the following activities. Clone the lab's code from GitHub using the following URL: https://github.com/cbourke/CSCE155-C-Lab05.

3.1 Using Functions

The file `orderStatistic.c` contains code necessary to find the i-th order statistic of a collection of numbers. The i-th order statistic corresponds to the i-th element in a sorted list. For example, the 4th order statistic of the list [5, 99, 23, 14, 6] is 23; the sorted list is [5, 6, 14, 23, 99], and 23 is the 4th element. Some special cases:

- The 1-th order statistic is the *minimum* element
- The n-th order statistic is the *maximum* element
- The $\frac{n}{2}$-th order statistic is the *median* element

The program converts command line arguments into i and an array of integers. It then sorts the array and outputs the i-th element. It does so by calling a series of functions.

Instructions

1. Manually compile the order statistics program:
   a) Compile the order statistics "library" using the following:
   `gcc -c -std=gnu99 orderStatistics.c`

   The `-c` flag tells gcc to compile (without linking) which produces an *object* file `orderStatistics.o`. This is not an executable program, but contains compiled code of the functions in `orderStatistics.c`

b) Compile your executable and *link* it to this "library" using the following:
`gcc orderStatistics.o orderStatisticsDemo.c`

2. Run the program with the following input values and verify that it is correct:
`./a.out 4 99 23 76 100 8 3 0 1 72 104 1000 12 18 14`

3. Complete questions 1 and 2 on your worksheet.

3.2 Writing Functions

Images are made up of individual *pixels*. Each pixel can be represented using an RGB (red-blue-green) color scheme. RGB is generally used in displays and models a color with three integer values in the range $[0, 255]$ corresponding to the red, green and blue "contribution" to the color. For example, the triple $(255, 255, 0)$ corresponds to a full red and green (additive) value which results in yellow.

Two common image filters are a black-and-white filter which transforms an image into an equivalent gray-scale image, and a sepia filter which transforms a photo to a reddish-brown monochrome to give it an old-timey look.

We have already written a substantial program that processes an image file and applies one of these filters. However, you will need to write the functions responsible for transforming an RGB value into a gray-scale or sepia RGB value.

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. Moreover, if any of the values exceeds 255, they should be reset to 255 to stay within the valid RGB color range.

Instructions

Design and write a "color utilities" library by implementing the functions described below. In particular:

- Place your prototypes and documentation in the file `colorUtils.h`
- Place your function definitions in the file `colorUtils.c`.
- Your color library functions are used in an image manipulation program (already written for you) that converts images to gray scale/sepia. To build your library and this program we've provided a `makefile` which defines the rules and dependencies required to build the project. From the command line, simply type `make` and it will produce an executable called `imageDriver`. Run the program to determine how to use it to test your functions.

We've provided starter code with two functions already completed for you. Implement the following functions:

- Write a helper function that returns the *minimum* of 3 integers. Name your function `min`.
- Write a function that takes three integer values (RGB values) and uses the lightness technique to return the gray-scale value. Name your function `toGrayScaleLightness`
- Write a function that takes three integer values (RGB values) and uses the luminosity technique to return the gray-scale value. Name your function `toGrayScaleLuminosity`
- Write three functions to compute the three sepia-tone RGB values. You'll need three functions because a function can only return one value. Name your functions `toSepiaRed`, `toSepiaGreen` and `toSepiaBlue` respectively.

3.3 Writing Tests

Testing is a fundamental step in software development. The most common types of tests are *unit tests* and the most common "unit" is a function. You may have already "tested" your functions by running the project and viewing the resulting images. However, for your final activity, you will write more automatable unit tests to test your functions more rigorously.

Some starter unit testing code has already been provided in `colorUtilsTester.c`. Each block of code calls a function with some inputs and compares the return value to the *expected* output. If they match, the test passes, if not it fails. A message is printed to the user and the total number of passed/failed test cases is reported.

Using this starter code as an example, add more test cases for the other functions. You should write at least 1 test case for *each* function you wrote.

To compile this tester you can use the same make file and the following:

```
make colorUtilsTester
```

which produces an executable program called `colorUtilsTester`

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.