EAST TENNESSEE STATE UNIVERSITY

CSCI-3160: Computer Systems

Data Lab

Note: this lab adapted from Bryant and O'Hallaron's *Computer Systems: A Programmer's Perspective (3E)***.** The acronym "CS:APP" refers to this text (our primary source for this course).

Goal and Setup

The purpose of this project is to help you become more familiar with C programming while further reinforcing bit-level representations of integers and floating-point numbers. After solving the puzzles presented within this assignment, you will (hopefully) have a better understanding of representing information as bits.

If you have not set up some sort of UNIX-like programming environment yet (Ubuntu via Windows Subsystem for Linux or Multipass, Cygwin, …), work on that first.

Windows users: I use the Windows Subsystem for Linux along with Ubuntu. See the instructions to set up WSL at https://docs.microsoft.com/en-us/windows/wsl/install.

Mac users: the "easiest" way to get a Linux environment running virtually on your laptop (if you don't want to use Clang in LLVM via XCode's development tools) will be using Canonical's Multipass: see https://multipass.run/. (Windows users: you could also use Multipass if you have any desire to muck about with learning about managing multiple VMs like you would in a cloud deployment.)

Once you have your UNIX-like programming environment, you may find yourself missing some utilities as you work through this lab. For Ubuntu/Debian users, familiarize yourself with the ***advanced package tool***, or ***apt***. See https://wiki.debian.org/Apt and https://manpages.ubuntu.com/manpages/bionic/man8/apt.8.html for more information.

Group Effort and Labor-Based Deliverables

I encourage students to form into small teams (for this assignment: pairs or groups of three would be best). Teammates are welcome to whiteboard ideas out and discuss approaches to solutions; however, I expect each student to code and submit their own solutions in bits.c in their individual GitHub Classroom repositories.

Given this project is like existing work from other institutions, you can (likely) find solutions to the problems floating around online or within the department. **Do not plagiarize, falsify, or otherwise cheat**: you rob yourself of the educational value (understanding information representation, more C practice) of this lab by copying answers from somewhere else.

This lab is labor-based: that means since this is introducing a lot of new concepts (C, git, GitHub and GitHub Classroom, Linux, WSL, …), I'm primarily interested in each student putting in an acceptable amount of effort towards completing the lab. To that end: your two deliverables for this lab are 1. Your bits.c file (the most recent version in your GitHub Classroom repository), and 2. A time.txt file added to your GitHub Classroom repository with an integer count of the number of hours you put in on this lab. **Count time associated with learning anything incidental to successful work on this lab!** That means learning about gcc, the Linux shell, git, GitHub, and any other pertinent topic or skill.

Lab Procedure

Getting Started

1. Download `datalab-handout` from your GitHub Classroom repository and copy it to a directory on your Linux VM that you plan to work out of (`/home/<your username>/datalab` is a good place to un-tar the handout in).

2. Run Visual Studio Code on your `datalab-handout` directory, or load the `bits.c` file in your favorite code editor—this is the **only** file you will modify.

The Puzzles

The `bits.c` file contains a skeleton for each programming puzzle you will solve. For each programming puzzle, you will complete each function scaffold using only ***straightline code*** (no loops or conditionals) and a limited number of C arithmetic and logic operators. Typically, you will be limited to the following operators:

```
! ~ & ^ | + << >>
```

A few of the functions further constrain the available operators you may use. In addition, you are only allowed to use constants no longer than 8 bits. See the comments above the function scaffolds in `bits.c` for detailed rules and examples of the desired coding style.

The table below (Table 1) lists the functions in a (rough) order of difficulty (easiest to hardest). The "Max Operators" column lists the upper limit of operators you can use to solve each puzzle. See `bits.c` for additional information and see `tests.c` for reference implementations of function behavior (note: the `tests.c` implementations do **not** satisfy the rules for solving the puzzles—that's the puzzle!)

Name	Description	Max Operators
`bitXor`	`x ^ y` using only `&` and `~`	14
`tmin`	Smallest two's complement integer	4
`isTmax`	True only if `x` is largest two's complement integer	10
`allOddBits`	True only if all odd-numbered bits in `x` set (equal 1)	12
`negate`	Return `-x` without using the `-` operator	5
`isAsciiDigit`	True if `0x30 ≤ x ≤ 0x39`	15
`conditional`	Same as `x ? y : z`	16
`isLessOrEqual`	True if `x ≤ y`, false otherwise	24
`logicalNeg`	Compute `!x` without using the `!` operator	12
`howManyBits`	Minimum number of bits to represent `x` in two's complement	90
`floatScale2`	Return bit-level equivalent of `2*f` for floating point argument	30
`floatFloat2Int`	Return bit-level equivalent of `(int)f` for floating point argument	30
`floatPower2`	Return bit-level equivalent of `2.0^x` for integer `x`	30

Table 1: Data Lab puzzles. The floating-point arguments are passed as unsigned integers (but assumed to have the same encoding as a 32-bit IEEE754 floating point number).

For the floating-point puzzles, you are allowed to use selection and repetition structures, along with larger-sized integer constants (both signed and unsigned). You are **not** allowed to use any data structures (unions, structs, or arrays) nor are you allowed to use any floating-point types, operations, or constants.

To help you understand the structure of floating-point numbers, you can use the included `fshow` program:

1. Compile `fshow`:

```
$ make
```

2. Run `fshow` to display what some arbitrary integer (encoded as a pattern of 1s and 0s) represents as an IEEE754 floating point number:

```
$ ./fshow 2080374784

Floating point value 2.658455992e+36
Bit Representation 0x7c000000, sign = 0, exponent = 0xf8, fraction = 0x000000
Normalized.  +1.0000000000 X 2^(121)
```

In addition, `fshow` will work with hexadecimal and floating-point values.

EVALUATION

I will be evaluating your solutions to the puzzles along the following criteria:

- *Correctness*: if your solution for a given puzzle passes the tests performed by btest (see below), you have a correct solution for that puzzle.
- *Performance*: I will check that your solution does not use more than the allotted operators using dlc (see below)—a correct and performant solution is "complete" from a grading perspective.
    - For those of you looking for an "above and beyond" opportunity: see if you can minimize the number of operations used. I will list the minimum number of operations for each puzzle found by all students in the class—being at the top (or near the top) of these lists is evidence for going above and beyond.
- *Style*: I will subjectively evaluate your solutions' style—aim for expressive, straightforward, clean code. Comments should detail **why** you did something, **not** what!

Note: this project is set up to award points based on correctness (1, 2, 3, or 4 points—see the comments in `bits.c` for how many points each exercise is awarded) and performance (2 points per function using up to the operator limit for that function). Given that point awards are "all or nothing"—this will be interpreted by me as checking off the correctness and performance of your puzzle solutions.

BTEST

This program checks the functional correctness in `bits.c`. Here's how you can build and use `btest`:

```
$ make
$ ./btest
```

Each time you modify your `bits.c` file, you will need to run make again to rebuild `btest`.

I suggest working through your functions one at a time, testing after each one as you go—you can use the `-f` option to instruct `btest` to only test a given function:

```
$ ./btest -f bitXor
```

You can also feed in function arguments from the command line using the `-1`, `-2`, and `-3` options:

```
$ ./btest -f bitXor -1 4 -2 5
```

See the included `README` file for additional `btest` documentation.

DLC

This program checks your puzzle solutions for compliance with the coding rules:

```
$ ./dlc bits.c
```

The program does not display any output unless it detects a problem. You can run `dlc` with the `-e` option to print counts of the number of operations used by each function. You can type `./dlc -help` for a list of command line options.

DRIVER.PL

This driver (tester external to the program—your `bits.c`) uses `btest` and `dlc` to determine the correctness and performance of your solution:

```
$ ./driver.pl
```

I will use this to check your puzzle solutions.

DELIVERABLES

Keep your `bits.c` source code file on your GitHub Classroom up to date with frequent commits. I will review the most recent pushed version of your puzzle solutions.