layout	title	subtitle	summary	date	released	due
lab	Data Lab	Looking at Types	This lab will help you become more comfortable and familiar with bit-level representations of integers and floating-point numbers. You will do this by solving a series of programming "puzzles."	2020-01-13 21:04:31 +0200	12:00 AM Tuesday, January 14th, 2020.	11:59 PM Friday, January 24th, 2020.

Weight: 5% of your total 50% Lab Grade

Note: The `conditional` function is no longer required.

Introduction

The purpose of this assignment is to become more familiar with bit-level representations of integers and floating point numbers. You'll do this by solving a series of programming "puzzles." Many of these puzzles may seem artificial, but you'll find yourself thinking much more about bits and encodings as you work through them. Also, bit manipulations are very useful in cryptography, data encoding, implementing file formats (e.g. MP3), and, although they shouldn't be done this way imho, certain job interviews.

Logistics

This is an *individual* project. All handins are electronic via Gradescope. Clarifications and corrections will be given during recitation sections. We strongly advise you to test your code on the `thoth` Linux machine before submitting it and to test your submission on Gradescope. We will not return any points lost because you did the lab in a different environment (even if it works there) if it does not work when tested on Gradescope before the deadline.

Remember that:

- You are not allowed to search for solutions online! (Although some resources on logical equivalence might be important refreshers)
- You are not allowed to ask other students for help, show them your code, or discuss the specifics of the solution.

Be aware that you may be asked to explain your code to a member of our course staff using only what you have submitted: your comments in the code should be such that you can determine what your code does and why a few weeks later, if needed. Write comments!!

Handout Instructions

First, log into the Linux Pitt machine and download the datalab package as follows:

```
wget https://bit.ly/2kB0Rvq -O datalab.zip
```

This will download the `datalab.zip` file to your (private) directory on the `thoth` Linux machine (make sure you are logged into this!) in which you will do your work. Then issue the command:

```
unzip datalab.zip
```

This will cause a number of files to be unpacked in the directory `datalab-handout`. Navigate to that directory and look at the files. The only file you will be modifying is `bits.c`. The `bits.c` file contains a skeleton for each of the programming puzzles. Your assignment is to complete each empty function using only the *straightline* code for the integer puzzles (i.e. no loops or conditionals) and a limited number of C arithmetic and logical operators. Specifically, you are *only* allowed to use the following eight operators:

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

A few of the puzzles further restrict this list. Also, you are not allowed to use any constants longer than 8 bits. See the comments in `bits.c` for detailed rules and a discussion of the desired coding style.

The Puzzles

This section describes the puzzles that you will be solving in `bits.c`. The following table describes a set of functions that manipulate the test sets of bits. The "Rating" field gives the difficulty rating (the number of points) for the puzzle. The "Max Ops" field gives the maximum number of operators you are allowed to use to implement each function. See the comments in `bits.c` for more details on the desired behavior of the functions. You may also refer to the test functions in `tests.c`. These are used as reference functions to express the correct behavior of your functions, although they do not satisfy the coding rules that constrain your own functions.

Name	Description	Rating	Max Ops
`bitXor(x,y)`	Return `x ^ y` using only `~` and `&`	1	14
`tmin()`	Return minimum two's complement integer	1	4
`allOddBits(x)`	Return 1 if all odd-numbered bits in word set to 1	2	12
`logicalNeg(x)`	Implement the `!` operator without using `!`	4	12
`floatFloat2Int(f)`	Return the bit-level equivalent of expression `(int)f` for floating point argument `f`	4	30

Some clarification on `floatFloat2Int`

The function `floatFloat2Int` implements a single-precision floating-point operation. The rules are greatly loosened for this puzzle. Here, you are allowed to use standard control-flow structures such as `if`, `for`, etc. You may use both `int` and `unsigned int` data types, including arbitrary unsigned and integer constants of any size. You may not use unions, structs, or arrays (nor would you need to.) Most significantly, you may not use any floating point data types, operations, or constants.

For this function, you will implement the casting operation `(int)f` which converts the floating point encoding to an integer encoding. This essentially means decoding the floating point value according to the IEEE 754 spec to pull out the integer value (and truncating away the fractional value.) In our slides, the process of encoding an integer value into the 32-bit floating point representation is shown. You are, effectively, performing the opposite operation.

When the value exceeds (is greater than INT_MAX) the possible integer range, you should return `0x80000000` which will represent an invalid integer for our purposes. Other values, such as NaN and Infinity of either sign should also result in `0x80000000`. Every other bit pattern will have a corresponding integer value that should be reflected in the result returned by your function.

You can use the provided `fshow` application to see the IEEE 754 floating-point bit pattern reflected by the given integer value. For instance:

```
thoth $ ./fshow 2080374784

Floating point value 2.658455992e+36
Bit Representation: 0x7c000000, sign = 0, exponent = f8, fraction = 000000
Normalized.  1.0000000000 X 2^(121)
```

You can also give `fshow` hexadecimal and floating point values which it will decipher in a similar way.

Evaluation

Your score will be computed out of a maximum of 15 points based on correctness. The puzzles you must solve have been given a difficulty rating between 1 and 4. We will evaluate your functions using the `btest` program, which is described in the next section. You will get full credit for a puzzle if it passes all of the tests performed by `btest`, and no credit otherwise.

Performance Points

Our main concern at this point in the course is that you can get the right answer. However, we want to instill in you a sense of keeping things consise yet legible. Finding such a sweet spot is a skill which you will become better at as you practice.

To that end, some of the puzzles can be accomplished by brute force or by some exhaustive logic, but we want you to be more clever. Thus, for each function we've established a maximum number of operators that you are allowed to use for each function. This limit is very generous and is designed only to catch the most egregiously inefficient solutions. You will receive **two points** for each correct function that satisfies the operator limit.

This is on top of the points earned for correctness alone that are based on the rated difficulty. A correct and performant solution to `floatFloat2Int`, for instance, will earn 4 + 2 points for a total of 6. Therefore, the total points for the lab are the 12 for correctness added to the 10 performance points for a total out of 22 points.

Auto-grading your work.

We have included some autograding tools in the handout directory: `btest`, `dlc`, and `driver.pl`.

btest: This program provides unit tests that check the functional correctness of the functions in `bits.c`. To build and use it, type the following two commands:

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

Notice that you must rebuild btest each time you modify your `bits.c` file. Just type the same two commands in when you make changes.

You'll find it useful to work through the functions one at a time and testing as you go. You can make use of the `-f` flag to instruct `btest` to test only a single function:

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

You can feed it specific function arguments by using the option flags `-1`, `-2`, and `-3` as necessary:

```
thoth $ ./btest -f bitXor -1 7 -2 0xf
```

Check the `README` for documentation on running the `btest` program.

dlc: This is a modified **ANSI C** (C89) compiler from the MIT CILK group that you can use to check for compliance with the coding rules for each puzzle. The typical usage is:

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

The program runs silently unless it detects a problem such as an illegal operator or too many operators. Running with the `-e` switch will make it noisy.

```
thoth $ ./dlc -e bits.c
```

This will have `dlc` print counts of the number of operators used by each function. Type `./dlc -help` for a list of command-line options.

driver.pl: This final script is a Perl script that will essentially just run the other two applications and compute your overall score. It takes no arguments.

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

We will use `driver.pl` to evaluate your solution.

Very Important Advice

- Just this once, do not include the `<stdio.h>` header in your `bits.c` file when you want to use `printf`. It confuses the `dlc` compiler and results in some strange errors. You will still be able to use `printf` in your `bits.c` file for debugging without including `<stdio.h>`. `gcc` will simply print some warnings you can ignore.
- The `dlc` program partly enforces the C89 dialect and not the more flexible C99. Specifically, it enforces the rule that variable declarations must be the first statements in the block. For example, it will complain about the following code:

```
int foo(int x) {
  int a = x;
  a *= 3;      /* Statement that is not a declaration */
  int b = a;   /* ERROR: Declaration not allowed here */
}
```

It does not seem to mind single-line comments using `//` however, which is nice.

Submission Instructions

You should hand in your `bits.c` solution file using Gradescope. If you're using a Shell Terminal, you can use `scp` to copy your file from the remote machine to your local machine as follows:

```
scp username@thoth.cs.pitt.edu:/path/to/remote/file /path/to/local/file
```

If you're using Windows, you can download the WinSCP software and enter with host `thoth.cs.pitt.edu` and then your username and password. Once in, locate your `bits.c` file and simply drag and drop to your Windows computer file system. There are also free GUI clients for Linux or Mac such as FileZilla (https://filezilla-project.org/).