1 Lab 4

Date: Sep 12, 2019

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

1.1 Aims

The aim of this lab is to introduce you to the limitations of computer arithmetic. After completing this lab, you should be familiar with the following topics:

- The approximation of integer aritmetic by modular arithmetic.
- Determining units in the last place for C `float`'s.
- Non-associativity of floating point arithmetic.
- *Type-punning* using C `union`'s.

1.2 Background

There are an infinite number of numbers. But computers (even with today's large memories) are finite. Hence it is impossible for any computer to represent all numbers and the numbers representable within computers are only a subset of the infinite set of all numbers.

Many programming languages provide two types of number representations:

Fixed-point representations This representation has a fixed maximum number of digits before and after the radix point. It has good precision but a limited range.

An example of a fixed-point representation are integers of various lengths where number of digits after the radix point is 0.

Floating-point representations The position of the radix point is not fixed but is specified by some kind of exponent (similar to *scientific notation* for decimal numbers). This representation allows a wider range but at the cost of a loss in precision.

1.3 Exercises

1.3.1 Starting up

Use the startup directions from the earlier labs to create a `lab4` directory and fire up a terminal whose output you are logging using the `script` command.

Make sure that your `lab4` directory contains a copy of the files directory.

You may need to evaluate the powers-of-2, so it may be a good idea to setup some kind of calculator to do so. A possible calculator is an interactive python shell started using `python` in a different terminal. Type `2**31` into it to evaluate 2-to-the-power-of-31.

1.3.2 Exercise 1: Unsigned Integer Overflow

Change over to the ex1 directory and look at the uints.c program. Once you have looked at the source code, build the `uints` executable by typing `make`. Run the program by typing `./uints`.

Type in interesting numbers which are on the 16-bit and 32-bit boundaries like `65535`, `65536`, `65537` and `4294967295`, `4294967296` and `4294967297`, and observe the results. Notice the silent overflow.

Terminate the program by typing `^D`.

1.3.3 Exercise 2: Signed Integer Overflow

Change over to the ex2 directory and look at the ints.c program. Once you have looked at the source code, build the `ints` executable by typing `make`. Run the program by typing `./ints`.

Type in interesting numbers which are on the 16-bit and 32-bit boundaries like `32767`, `32768`, `32769` and `65535`, `2147483647`, `2147483648`, `2147483649`, `42¬94967295` and observe the results. Notice the silent overflow with changing sign.

Terminate the program by typing `^D`.

1.3.4 Exercise 3: Identifying a Mask

This exercise requires you to use what you learnt from the previous couple of exercises to identify a mask. Change over to the ex3 directory and look at the source files contained there. You will notice that the directory contains an object file `mystery.o` without any corresponding source file. The code in `mystery.o` contains a function `mystery()` which returns only the lowest `n`-bits of its argument. Your task is to figure out `n` by using the `identify` program.

Build the program by typing `make` and then run it by using `./identify`. You can type in integers in hexadecimal (without any leading `0x`). The program outputs the result of the `mystery()` function on each input integer.

Come up with suitable input which allows you to figure out how many lowest bits of its argument are returned by the `mystery()` function.

1.3.5 Exercise 4: An Integer equal to Its Negation

Change into the ex4 directory and type `make`. Then run the program `./negeq`. This program requires you to input an `int` which is equal to its own negation. Like the previous exercise, the program takes its input in hex (without any leading `0x`). You should provide some integer `n` in hex such that the program outputs `N == -N` where `N` is the decimal representation of hex integer `n`.

Consider the previous exercises to identify this value (Hint: recall the asymmetry of 2's complement).

1.3.6 Exercise 5: Infinite Precision Integers

As the previous exercises illustrate, in C integers are represented with a small precision like 16 or 64 bits. Newer languages like Python, Ruby, Perl6 allow integers with precision limited only by available memory.

For example, fire up an interactive python using `python` and type `10**1000 - 1`. You should see output containing quite a few lines of `9`'s. It's kind of neat that such examples work (some languages like Scheme even allow rational numbers where numbers are represented as fractions with fraction arithmetic).

However, now within `irb`, type `10**1000 - 1.0`. You should get back a message which should reveal to you that the representation of numbers within computers is only an approximation of numbers.

1.3.7 Exercise 6: 0.1 cannot be represented

Recall from class that the floating point representation commonly used with present day computers is a binary representation. That means that numbers which can be represented by fractions with denominators which are a power-of-2 can be represented exactly but other numbers cannot. For example, 0.1 which is 1/10 cannot be represented exactly and this exercise illustrates this problem.

Change over to the ex6 directory and look at the code in 0.1.c You should see that all it is doing is summing up 0.1 10 times, printing out the resulting sum as well as checking whether the sum is equal to 1.

Build and run the program and observe the result.

1.3.8 Exercise 7: A Number Not Equal To Itself

Change over to the ex7 directory. The program in nan.c requires you to enter a number, divides that number by itself to get `x` and then loops as long as `x != x`. Compile and run the program and then provide an input to force the program into an infinite loop. Hint: The number `NaN` is not equal to itself, hence if your input sets `x` to `NaN` the program will loop.

Once you are successful, terminate the infinite loop by typing a `control-C`.

1.3.9 Exercise 8: Unit in Last Position

This exercise illustrates that the value of the *unit in last place* of a floating point number increases dramatically with the value being represented.

Change into the ex8 directory and build the `ulp` program by typing `make`. Then run `./ulp`, it should output a usage message. Run it as `./ulp verbose` and it should produce a dump on standard output of the value of a single-precision `float` number being represented along with the value of the unit in the last place of the number. Note that as the magnitude of the values increase, the ULP rapidly goes above 1. That means those values cannot be distinguished even if they differ by 1.

It may be a good idea to look at the ULP distribution on a graph. Run `./ulp data >ulp.data` which dumps out the ULP distribution in a format acceptable to the `gnuplot` plotting program. Display the graph using `gnuplot -p ulp.gp`. Unfortunately, because of the linear scale, most of the values are concentrated near the origin.

This cries out for the use of logarithmic scales. Run `./ulp lg-data >ulp-¬ lg.data` which dumps out the log (base-2) of the data into `ulp-lg.data`. Now display the graph using `gnuplot -p ulp-lg.gp`. This should produce a much cleaner plot.

It is worth understanding how the ulp program works. The key to its working is the `FloatInt` `union`. A `union` is similar to a `struct` except that its members occupy the same memory location. Hence it is possible to use `union`'s to get at the representation of types as is done here.

The floating point member of the union is assigned a power-of-2. That means that the normalized floating point mantissa will be all zeros. Thus by adding 1 to the integer member of the union (recollect that the integer and float members occupy the same memory), we are incrementing the ULP of the floating point member. Hence the value of the ULP is the value of the incremented member minus the original value.

1.4 Exercise 9: Different Numbers are not Distinguishable

Two floating point numbers which differ by less than the ULP for their values cannot be distinguished.

Change into the ex9 directory and build and compile loop.c. Look at the code in loop.c, and based on the results of the previous exercise, provide a minimal power-of-2 input which causes it to go into an infinite loop.

1.5 References

Text, Ch. 2.

Joshua Bloch and Neal Gafter, *Java Puzzlers: Traps, Pitfalls and Corner Cases*, Addison-Wesley, 2005. Source of some of the exercises.