Lab 1: Bits, bytes, and integers

Lab sessions Tue Oct 03 to Thu Oct 05

Lab written by Julie Zelenski

How lab works

Your weekly lab is a chance to experiment and explore, ask and answer questions, and get hands-on practice in a supported environment. We provide a set of lab exercises that revisit topics from recent lectures/readings and prepare you to succeed at the upcoming assignment.

Lab is collaborative! You will pair up and work as a team on the exercises. The entire room is one learning community working together to advance the knowledge and mastery of everyone. Stuck on an issue? Ask for help. Have an insight? Please share! The TA will circulate to offer advice and answers and keep everyone progressing smoothly.

To track lab participation, we have an online checkoff form for you to fill out as you work. Lab is not a race to find answers to exactly and only the checkoff questions-- the checkoff questions are intentionally simple and used only to record attendance and get a read on how far you got. Lab credit is awarded based on your sincere participation for the full lab period. Your other rewards for investing in lab are to further practice your skills, work together to resolve open questions, satisfy your curiosity, and reach a place understanding and mastery. The combination of active exploration, give and take with your peers, and the guidance of the TA makes lab time awesome. Hope you enjoy it!

Learning goals

During this lab you will:

1. practice with bits, bitwise operators and bitmasks
2. read and analyze C code that manipulates bits/ints
3. work through the edit-compile-test-debug cycle in the unix environment

Find an open computer to share with a partner and introduce yourselves. Together the two of you will tackle the exercises below.

Get started

Clone the lab repo by using the command below. This command creates a lab1 directory containing the project files.

```
git clone /afs/ir/class/cs107/repos/lab1/shared lab1
```

Pull up the online lab checkoff (https://web.stanford.edu/class/cs107/cgi-bin/lab1) right now and have it open in a browser so you can jot things down as you go. Only one checkoff needs to submitted for both you and your partner.

Lab exercises

Let's kick things off with a little **unix love**! Chat up your labmates about your assign0 experiences. How is everyone doing so far on getting comfortable in the unix environment? Do you have open questions or an issue you'd like help with? Did you learn a nifty trick or pro tip that you'd like share? Let's hear it!

1) Bitwise pratice

Test your understanding of the recent lecture material with this page of bitwise practice problems (practice.html).

2) Code to read and learn from

This quarter we are mining our favorite open-source projects (musl libc (http://www.musl-libc.org), BusyBox unix utilities (https://busybox.net/about.html), Apple (https://opensource.apple.com), Google (https://opensource.google.com) and more) for example systems code to use as an object of study. We will use this code to learn how various programming techniques are used in context, give insight into industry best practices, and provide opportunities for reflection and critique.

For lab1, we plucked three passages that highlight interesting uses of the bitwise operations for you to consider. This code is also repeated in `code.c` so you can execute it and examine under the debugger.

Round up

Functions like `roundup` shown below are commonplace in systems code. The function returns the value of the first argument rounded up to the nearest multiple of the second. Its clever use of bitwise operations avoids the more expensive multiply/divide instructions:

```
// NOTE: only works correctly if mult argument is exact power of two
size_t roundup(size_t sz, size_t mult)
{
    return (sz + mult-1) & ~(mult-1);
}
```

Questions for you to ponder:

1. What is the return value of `roundup(9, 2)` ? of `roundup(33, 32)` ? of `roundup(4, 4)` ?
2. What is a `size_t` ? Why does this code use `size_t` instead of `int` type?
3. Trace through the function's operation and explain how the function works.
4. The comment says the function requires that `mult` be a power of two. What does the function return when `mult` is not a power of two?

I found a similar code passage to round up a size in BusyBox, which I paraphrase below:

```
return (sz + mult) & -mult;
```

This version has two differences from the previous: it has substituted `-mult` for what was `~(mult-1)` and is missing a `-1` term in the addition.

1. The first change has no effect on behavior/result. Why not?
2. The second does have a consequence. Identify in what situations it affects the rounded result and why. (Is this intentional or an error/oversight? I think the latter...)

Absolute value

A straightforward implementation of absolute value looks like this:

```
int abs_val(int x)
{
    return (x < 0 ? -x : x);
}
```

(Aside: note the use of C's `?:` operator which is like a compact if/else expression). The code above is simple and obviously correct, so it's unclear why someone went to the trouble to write this version:

```
int abs_val(int x)
{
    int sn = x >> (sizeof(int)*CHAR_BIT -1);
    return (x ^ sn) - sn;
}
```

Huddle with your partner to puzzle it out on paper to see how this version works -- it may take some head scratching! It will help to know that the `sizeof` operator reports the number of bytes for a type and that the `CHAR_BIT` constant is the count of bits in a byte (which is near universally 8, but apparently someone was ultra-cautious about making assumptions...)

This seemingly roundabout computation is used here because of its performance advantages. A characteristic of many modern systems is to strongly reward code that follows a straight-line path (i.e. control flow does not divide into separate paths, no test/branch). Bitwise tricks are often employed to rework branchy code into a branchless form for the performance advantage. Interesting!

Side notes: for abs_val to work correctly, right-shift on a signed value must be **arithmetic** to replicate sign bit of a negative value; it will fail if signed right-shift is **logical** and zero-fills. It is typical for signed right-shift to be arithmetic (and is true for myth/gcc), but the C standard does not require it, so depending on such behavior is not portable to all systems. The code presented above was patented by Sun Microsystems in 2000 (https://www.google.com/patents/US6073150) but I hear that rearranging into the functionally equivalent `(x + sn) ^ sn` may be enough to avoid being hunted down by a mob of IP lawyers.

Min

The `min` function below is thematically similar to branchless absolute value.

```
int min(int x, int y)
{
    int diff = x-y;
    return y + (diff & (diff>>31));
}
```

First trace through how it works:

1. First line is ordinary subtraction.
2. Consider two cases: `diff` is negative (y is larger than x) or `diff` positive (y is smaller/equal to x).
3. `diff` positive: `diff>>31` is 0, &'ed against `diff` equals 0, `y` + 0 is just `y`.
4. `diff` negative: `diff>>31` is -1, &'ed against `diff` equals `diff`, `y + diff` gets you back to `x`

Okay, we're convinced it works for the ordinary cases, but what about at the extremes? Consider in what situations the subtraction can overflow beyond `INT_MIN` or `INT_MAX`. For what sort of values `x` and `y` does this happen? What happens at runtime when an integer operation overflows? Not sure? Try running the code to see. What is the result of the function in the case when the subtraction overflows?

A branchless min that uses a few more instructions (but no subtraction and no concern about overflow) is shown below. When you have some free time (after lab!) see if you can work out how this one works!

```
int min(int x, int y)
{
    return y ^ ((x ^ y) & -(x < y));
}
```

3) Write, test, debug, repeat

Now it's your turn to work up some bitwise code of your own and practice with the unix development tools!

The `parity` program reports the *parity* of its command-line argument. A value has odd parity if there is an odd number of "on" bits in the value, and even parity otherwise. Confirm your understanding of parity by running the `samples/parity_soln` program on various arguments.

The code in `parity.c` was written by your colleague who claimed it is "complete", but on his way out the door he mutters something unintelligible about unfixed bugs. Uh oh... Your task is to test and debug the program into a fully functional state using CS107 sanitycheck and the `gdb` debugger.

parityA. This function was attempted by your colleague, then abandoned when he couldn't get it working. Let's investigate!

1. Use `make` to build the program and try running `parity` a few times on various values. Uh, it thinks every value has odd parity? Does the program ever report even parity for anything?
2. Let's get it under the debugger. Start `gdb parity`. Use `list parityA` to print the function and set a breakpoint on the line number inside the `if` where it toggles parity for an "on" bit.
3. Run the program under gdb and when first stopped at the breakpoint, print the value of `parity` before toggling. Huh? The value of `parity` it total garbage- d'oh, it was never initialized! Is that even legal? In a safety-conscious language such as Java, the compiler outright rejects code that uses an uninitialized value.
4. Do a `make clean` and `make` to review the build warnings and you'll see nary a peep about it from gcc. At runtime, the variable will use whatever junk value was leftover in its memory location. Lesson learned -- you will need to up your own vigilance in the laissez-faire world of C.
5. Add a correct initialization, build, and re-run to test your fix.

Sanitycheck! One simple means to verify correctness is by comparing your results to a known-correct solution. We provide solution executables in the samples directory. For example, run `./parity 45` then run `samples/parity_soln 45` and manually eyeball the outputs to confirm they match. Even better would be to capture those outputs and feed them to `diff` so the tools can do the work. To make testing as painless as possible for you, we've automated simple output-

based comparison into a CS107 tool called `sanitycheck`. If you haven't already, read our sanitycheck instructions (/class/cs107/sanitycheck.html), and run sanitycheck on lab1 to see that your fixed parityA passes all tests.

parityB. Your colleague says this version is ready to ship. Do you agree?

1. Edit `main` to call `parityB` instead of `parityA`. Re-build and test by running default sanitycheck -- it passes. Good to go, right? Not so fast, keep in mind that sanitycheck is only as thorough as its test cases...
2. Run custom sanitycheck with the additional test cases given in `custom_tests` to get a different side of the story. It times out on one of these tests. It's not that parityB is horribly inefficient, the program is stuck an infinite loop.
3. You need to get the program back under the debugger to get visibility on what's happening. Once under the debugger, run the parity program on a negative argument and let it go unresponsive.
4. Type Control-C to interrupt it and return control to gdb. Use the gdb `backtrace` command to see where the program is executing. `step` through a few statements as it goes around the loop and gather information to diagnosis why the loop is not being properly exited.
5. Once you know what's gone astray, edit the code to fix, rebuild, and test under both default and custom sanitycheck to verify you have squashed the bug. Way to go!

Check off with TA

At the end of the lab period, submit the checkoff form and ask your lab TA to approve your submission so you are properly credited for your work. It's okay if you don't completely finish all of the exercises during lab; your sincere participation for the full lab period is sufficient for credit. However, we highly encourage you to finish on your own whatever is need to solidify your knowledge. Try our self-check (/class/cs107/selfcheck.html#lab1) to reflect on what you got what from this lab and whether you feel ready for what comes next!

Just for fun

- a cute parlor trick (binarycards.html) based on integer representation
- a surprisingly addictive binary Tetris game (https://studio.code.org/projects/applab/iukLbcDnzqgoxuu810unLw)
- crazy bit-hacks (http://graphics.stanford.edu/~seander/bithacks.html) for the truly brave.
- integer overflow wrecking worldwide havoc: Nuclear Gandhi (https://www.geek.com/games/why-gandhi-is-always-a-warmongering-jerk-in-civilization-1608515/), grounding the Boeing Dreamliner (https://www.engadget.com/2015/05/01/boeing-787-dreamliner-software-bug/), Gangnam Style (http://www.exploringbinary.com/gangnam-style-video-overflows-youtube-counter/) breaks the internets

---