Assignment 1: A bit of fun

Due: Mon Oct 9 11:59 pm
Ontime bonus 5%. Grace period for late submissions until Wed Oct 11 11:59 pm

Assignment by Julie Zelenski

Learning goals

This assignment delves into those topics covered in the first week of lecture and explored in lab1 (/class/cs107/lab1/). It is designed to give you

- first experiences editing, compiling, testing, and debugging C programs under Unix
- reading and analyzing code that manipulates bits and integers
- writing code using the C bitwise operators
- exposure to working within/around the limits of integer representation

Overview

Watch video walkthrough! (https://www.youtube.com/watch?v=8WUvv4sYrGU)

Your first assignment is designed as a "programming problem set", with a mix of code-reading and code-writing tasks.

For each code-reading exercise, you are asked to read and analyze a passage of code and answer a few targeted questions concerning it. Your answers are to be typed into your readme.txt file.

For the code-writing problems, we provide a partially-written program and you are tasked with implementing a function or two to bring the program to life. The code we provide covers all the mundane scaffolding, allowing your efforts to focus on the interesting manipulation of bits and integers.

The code for each problem, whether reading or writing, is quite short, but these small passages are mighty! Digging into this code will solidify your understanding of bits, integer representation, and computer arithmetic.

Note: Although we refer to this as problem set and the exercises are similar in spirit to the partnered explorations you do in lab, we want to be very clear about the collaboration restrictions (/class/cs107/collaboration.html) for assignments. When working on any CS107 assignment, you are expected to do your own independent thinking, design, coding, and debugging. Discussions of the assignment with other students are okay at a very high-level, but should never descend into detailed discussions nor exchanging solutions and code-level specifics. If you need assignment-specific help, please reach out to the course staff!

Get started

Check out the starter project from your cs107 repo using the command

```
git clone /afs/ir/class/cs107/repos/assign1/$USER assign1
```

The starter project contains a `code.c` file with the code for the code-reading exercises, three partially-written programs (`sat.c` , `automata.c` and `utf8.c`) and the supporting `Makefile` , `custom_tests` , and `readme.txt` files. In the `samples` subdirectory, you'll find our sample solutions.

All of the programs are designed to be conveniently tested with our sanitycheck (/class/cs107/sanitycheck.html) tool.

1. The power of two

In the myth system header file `<sys/param.h>` (full path `/usr/include/sys/param.h`) you'll find the following helper (excuse the quirky syntax due to being defined as a preprocessor macro):

```
#define powerof2(x)     ((((x) - 1) & (x)) == 0)
```

The intent is for `powerof2(x)` to evaluate to true when x is a power of 2 and false otherwise. Work out for yourself how `powerof2` operates and then answer the following questions. Type your answers in your readme.txt file.

a. First just consider positive numbers. Jot down the binary representation for the first few powers of 2 (e.g. 1, 2, 4, 8, ...) and compare to the binary representations of non-powers such as 5 and 14. What property of the bit pattern of an unsigned int holds true if and only if the value is an exact power of two?
b. There is exactly one negative signed int in two's-complement representation which has the above property. What value is it?
c. Briefly explain the bitwise/numeric mechanism by which `powerof2` detects an exact power of two.
d. What is the result from `powerof2` when evaluated on the negative value from your answer to (b)?

2. Finding the middle ground

Who knew that computing the midpoint of two numbers could be perilous? Read Google researcher Joshua Block's newsflash about an overflow bug (https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html) and ponder how it took 20+ years for this ubiquitous bug to finally surface.

We want a function that safely and correctly computes the midpoint of two integers. If the actual midpoint is not an integer, we are not fussy about whether the function rounds up, down, or toward zero, either neighbor is acceptable. The `mid_orig` function shown below mostly works, but exhibits the overflow bug called out in Block's article:

```
int mid_orig(int low, int high)
{
    return (low + high)/2;
}
```

If the sum `low + high` overflows, the result is erroneous. For example, the call `mid_orig(INT_MAX-2, INT_MAX)` should return `INT_MAX-1`, but actually returns -2. Oops!

Three proposed alternative midpoint functions are given below. Block offers the fix shown in `mid_A` and champions `mid_B` as a faster alternative. `mid_C` is just one of the many gems in the fascicle on bits (http://proquest.safaribooksonline.com/9780321637413?uicode=stanford) written by the extraordinary Don Knuth. (fascicle??).

```
int mid_A(int low, int high)
{
    return low + ((high - low) / 2);
}

int mid_B(int low, int high)
{
    return ((unsigned int)low + (unsigned int)high) >> 1;
}

int mid_C(int low, int high)
{
    return (low & high) + ((low ^ high) >> 1);
}
```

Answer the following questions. Type your answers in your readme.txt file.

a. Both `mid_A` and `mid_B` work correctly **as long as both inputs are non-negative**. This constraint is never clearly stated, just implicit in the original context that low and high are array indexes. Things may not be quite so rosy if one or both inputs is negative. Unambiguously identify in which situations a negative input causes trouble for `mid_A` and demonstrate with a specific call that returns an incorrect result.
b. `mid_B` also has its own problem with negative inputs. Unambiguously identify in which situations negative inputs are trouble for `mid_B` and demonstrate with a specific call that returns an incorrect result.
c. Knuth's `mid_C` is the real deal, correct result for all inputs, no possibility of overflow! How this approach works is not at all obvious as first glance, so invest some quality time to work through the bitwise manipulation and integer representation until you see how it all fits together. *Hint:* consider the bitwise representation and its relationship to a binary polynomial/powers of 2. Once you have it figured out, write a short explanation of how and why it works. Your answer may focus solely on the case where both inputs are non-negative inputs. (While it is possible to generalize to the case when one or more inputs is negative, it is trickier to reason about.)

d. Including an optional haiku about bitwise magic and the genius of Don Knuth will earn a smile from your grader. We'll post the best ones to the forum!

3. Ground zero

The code below was taken from musl (http://www.musl-libc.org) and lightly paraphrased. The task is to determine whether any of the 8 bytes in a long is a zero byte. The simple and naive approach would mask out each byte and test it individually. The function below manages to simultaneously test all bytes. What?! Puzzling out the operation of this extraordinary function is going to take some effort, but I promise it will be worth it!

```
bool has_zero_byte(unsigned long val)
{
    unsigned long ones = ~0UL/UCHAR_MAX;
    unsigned long highs = ones << (CHAR_BIT - 1);
    return ((val - ones) & (~val & highs)) != 0;
}
```

Answer the following questions. Type your answers in your readme.txt file.

a. Describe the bitmask constructed by the expression `~0UL/UCHAR_MAX`. The suffixes on the constant `0UL` are critical. Explain the effect on the expression if the `L` were removed. Now explain the effect of removing the `U`.
b. The original version of the code expressed the second line as:

   ```
   unsigned long high = (ones * (UCHAR_MAX/2+1))
   ```

   Demonstrate why the original and replacement expressions are equivalent.
c. Describe what is computed by the expression `val - ones`.
d. Describe what is computed by the expression `~val & highs`.
e. Now take it over the finish line: explain how combining those two expressions produces the desired result.

The function demonstrates a form of "mini-parallelism"-- a single operation on the entire long is effectively applying that operation to all 8 bytes in parallel. While the naive loop racks up 3-6 operations per **each** byte (a total of some 40 ops), this version tests **all** bytes in just 4 total operations (assuming constant operations are precomputed at compile-time). Neat! This is an exceedingly clever piece of code and I hope a satisfying accomplishment for you to be able to understand and appreciate such an action-packed sequence of bitwise wizardry!

Review and comment starter code

When you inherit code as starting point, such as for this assignment, your first task should be to **carefully review it**. You want to know how it operates and what it does and doesn't handle so you can plan how the code you write will fit into the program. For this assignment, there is more code in the starter than you will write yourself. You are given about 20 lines in each program and you will add just 10 more lines of your own!

For this assignment, the code we provide processes the command-line arguments and handles user error. This kind of code is fairly rote and can be somewhat goopy. One idiomatic pattern to note is how to safely convert string arguments to numbers. Assign0 used the venerable `atoi` function which is simple to use but provides zero error-detection. Switching to the full-featured `strtol` adds flexibility and robustness to the conversion, but requires more complex code. Read the function documentation at `man strtol` and review how the function is used in `convert_arg`.

Some questions you might consider for self-test: (do not submit answers)

- How is the second argument to `strtol` used for error-detection? If you don't want that error-detection, what do you pass as the second argument instead?
- When converting the user's string argument to a number, what base is used?
- What is the `error` function and how is it used?
- How do the sample programs respond if invoked with missing arguments? excess arguments?
- Do you see anything unexpected or erroneous? *We intend for our code to be bug-free; if you find otherwise, please let us know!*

We intentionally left off all the comments on the starter code and expect you to pick up the slack and add the missing documentation. You should add an appropriate overview comment for the entire program, document each of the functions, and add inline comments where necessary to highlight any particularly tricky details.

4. Saturating arithmetic

The `sat` program performs a synthetic saturating addition operation. *Saturating* addition clamps the result into the representable range. Instead of overflowing with wraparound as ordinary two's-complement addition does, a saturating addition returns the type's maximum value when there would be positive overflow, and minimum when there would be negative overflow. Saturating arithmetic is a common feature in 3D graphics and digital signal processing applications.

Below shows two sample runs of the `sat` program:

```
$ ./sat 8
8-bit signed integer range
min: -128   0xffffffffffffff80
max:  127   0x000000000000007f
$ ./sat 8 126 5
126 + 5 = 127
```

The program reports that an 8-bit signed value has a range of -128 to 127 and if you attempt to add 126 to 5, the result overflows and sticks at the maximum value of 127.

You are to implement the functions below to support saturating addition for the `sat` program.

```
long signed_min(int bitwidth);
long signed_max(int bitwidth);
long sat_add(long operand1, long operand2, int bitwidth);
```

The `bitwidth` argument is a number between 4 and 64. A two's-complement signed value with `bitwidth` total bits is capable of representing a fixed range of values. The `signed_min` and `signed_max` functions return the smallest and largest values of that range. The `sat_add` function implements a saturating addition operation which returns the sum of its operands if the sum is within range or the appropriate min/max value when the result overflows. The type of the two operands is `long` but you can assume the value of the operands will always be within the representable range for a `bitwidth` -sized signed value.

There are two important restrictions on the code you write for this problem:

- **No relational operators**. Your code is prohibited from making any use of the relational operators. This means absolutely no use of `< > <= >=` .
- **No special cases based on bitwidth.** Whether the value of bitwidth is 4, 64, or something in between, your functions must use one unified code path to handle any/all values of bitwidth without special-case handling. No use of `if switch ?:` to divide the code into different cases based on the value of bitwidth. This doesn't mean that you can't use conditional logic (such as to separately handle overflow or non-overflow cases), but conditionals that dispatch based on the value of bitwidth or make a special case out of one or more bitwidths are disallowed.

A solution that violates either restriction will not receive credit, so please verify your approach is in compliance!

You will want to test your saturating addition on a wide variety of inputs. This means different bitwidths and sums in range and those that overflow. If you use custom sanitycheck (/class/cs107/sanitycheck.html) for your tests, you'll get the benefit of having automatic comparison to our sample solution, very convenient!

5. Cellular automata

Many of you implemented Conway's delightful Life simulation as a CS106B assignment and this problem will revisit cellular automaton, albeit in a simpler form and implemented via bit tricks. Daniel Shiffman's neat book *The Nature of Code* contains an excellent explanation of the elementary cellular automaton. Rather than have me attempt to recreate it here, please read Sections 7.1 and 7.2 (http://natureofcode.com/book/chapter-7-cellular-automata/) for essential background information. Section 7.3 goes on to develop an automata program, but it employs a more heavyweight array/OO design that we eschew in favor of a low-level bitwise approach.

We compactly represent a generation as a 64-bit unsigned long, one bit for each cell. A 1 bit indicates the cell is live, 0 if not. Using this bit-packed representation, the code to read or update a cell is implemented as a bitwise operation.

Let's trace how one generation advances to the next. A cell and its two neighbors form a "neighborhood". A neighborhood is effectively a 3-bit number, a value from 0 to 7. Consider a live cell with a live left neighbor and an empty right neighbor. The cell's neighborhood in binary is `110`, which is the 3-bit number 6. The ruleset dictates whether a cell with neighborhood 6 will be live or empty in the next generation. A ruleset is represented as an 8-bit number, one bit for each of the 8 possible neighborhood configurations. Let's say the ruleset is 93. The number 93 expressed in binary is `01011101`. Because the bit at position 6 (note: position 0 -> least significant) of the ruleset is 1, this cell will be live in the next generation. Are you getting the sense there will be much binary data to practice your bitwise ops on for this problem?

The `automata` program animates an elementary cellular automaton starting from the initial generation and applying a specified ruleset. Below shows a sample run of the program for ruleset 93:

```
$ ./automata 93
                               +
++++++++++++++++++++++++++++++ +++++++++++++++++++++++++++++++++
+                            + +                               +
++++++++++++++++++++++++++++ + +++++++++++++++++++++++++++++++ +
+                          + + +                             + +
++++++++++++++++++++++++++ + + +++++++++++++++++++++++++++++ + +
+                        + + + +                           + + +
++++++++++++++++++++++++ + + + +++++++++++++++++++++++++++ + + +
+                      + + + + +                         + + + +
++++++++++++++++++++++ + + + + +++++++++++++++++++++++++ + + + +
+                    + + + + + +                       + + + + +
++++++++++++++++++++ + + + + + +++++++++++++++++++++++ + + + + +
+                  + + + + + + +                     + + + + + +
++++++++++++++++++ + + + + + + +++++++++++++++++++++ + + + + + +
+                + + + + + + + +                   + + + + + + +
++++++++++++++++ + + + + + + + +++++++++++++++++++ + + + + + + +
+              + + + + + + + + +                 + + + + + + + +
++++++++++++++ + + + + + + + + +++++++++++++++++ + + + + + + + +
+            + + + + + + + + + +               + + + + + + + + +
++++++++++++ + + + + + + + + + +++++++++++++++ + + + + + + + + +
+          + + + + + + + + + + +             + + + + + + + + + +
++++++++++ + + + + + + + + + + +++++++++++++ + + + + + + + + + +
+        + + + + + + + + + + + +           + + + + + + + + + + +
++++++++ + + + + + + + + + + + +++++++++++ + + + + + + + + + + +
+      + + + + + + + + + + + + +         + + + + + + + + + + + +
++++++ + + + + + + + + + + + + +++++++++ + + + + + + + + + + + +
+    + + + + + + + + + + + + + +       + + + + + + + + + + + + +
++++ + + + + + + + + + + + + + +++++++ + + + + + + + + + + + + +
+  + + + + + + + + + + + + + + +     + + + + + + + + + + + + + +
++ + + + + + + + + + + + + + + +++++ + + + + + + + + + + + + + +
++ + + + + + + + + + + + + + + +   + + + + + + + + + + + + + + +
++ + + + + + + + + + + + + + + +++ + + + + + + + + + + + + + + +
```

You are to implement these functions for the `automata` program.

```
void draw_generation(unsigned long gen);
unsigned long advance(unsigned long gen, unsigned char ruleset);
```

The `draw_generation` function outputs a row of 64 characters, one for each cell. A live cell is shown as an inverted box, an empty cell is a blank space. The generation is ordered such that the cell corresponding to the most significant bit is in the leftmost column and the least significant bit in the rightmost column.

The `advance` function advances the generation forward one iteration. For each cell, it examines its neighborhood and uses the ruleset to determine the cell's state in the next generation. The two cells at the outside edges require a small special case due to having only one neighbor; their non-existent neighbor should be treated as though permanently off.

When testing your program, try playing with different values for the rulesets. Can you find a ruleset that produces Sierpinski's triangle? What about an inverted Sierpinski? Which ruleset produces your most favorite output?

Rather than trying to manually eyeball the output for correctness, sanitycheck (/class/cs107/sanitycheck.html) can be used to compare your output to that of the sample. Sanitycheck converts the output to show live cells as `#` and empty as `.`. This makes the output less aesthetically pleasing, but easier for visual discrimination.

6. UTF-8

A C char is a one-byte data type, capable of storing 2^8 different bit patterns. The ASCII standard (`man ascii`) establishes a mapping for those patterns, but limited to 256 options, only ordinary letters, digits, and punctuation make the cut to be included. Unicode is ASCII's more cosmopolitan cousin, defining a universal character set that supports glyphs from a wide variety of world languages, both modern and ancient (Egyptian hieroglyphics, the original emoji? (http://www.unicode.org/charts/PDF/U13000.pdf)).

A Unicode character is identified by a number called its *code point* and the Unicode codespace encompasses over a million different code points. The notation U+NNNN refers to the code point whose value is hex 0xNNNN. The `utf8` program takes one or more code points and displays its Unicode character. Here is a sample run:

```
$ utf8 0x41 0xfc 0x3c0 0x4e8a
U+0041   Hex: 41         Character: A
U+00FC   Hex: c3 bc      Character: ü
U+03C0   Hex: cf 80      Character: π
U+4E8A   Hex: e4 ba 8a   Character: 亊
```

UTF-8 is an encoding used for Unicode. It represents a code point as a sequence of bytes; the length of the sequence is determined by the number of significant bits in the code point's binary representation. The following table shows the structure for one, two and three-byte UTF-8 encodings. If the number of significant bits in the code point is no more than 7, it fits into the one-byte sequence; if number of significant bits is between 8 and 11, it requires a two-byte sequence, and if between 12 and 16 bits, a three-byte sequence. (There is also a four-byte encoding, but we will ignore it).

Code point range	**Num sigbits**	**UTF-8 encoded bytes (in binary)**
U+0000 to U+007F	7	`0xxxxxxx`
U+0080 to U+07FF	11	`110xxxxx 10xxxxxx`
U+0800 to U+FFFF	16	`1110xxxx 10xxxxxx 10xxxxxx`

The `0` and `1` bits shown in the UTF-8 bytes above are fixed for all code points. The `xxx` bits store the binary representation of the code point. The one-byte sequence stores 7 bits, the two-byte sequence stores 11 bits (5 bits in first byte and 6 in the other), and the three-byte stores 16 bits (divided 4, 6, and 6).

In a single-byte sequence, the high-order bit is always `0` , the other 7 bits are the value of the code point itself. (The first 128 Unicode code points deliberately use the same single byte representation as the corresponding ASCII char, for backwards-compatibility with older systems that only know about ASCII -- neat!)

For the multi-byte sequences, the first byte is called the *leading* byte, the subsequent byte(s) are *continuation* bytes. The high-order bits of the leading byte indicate the number of bytes in the sequence. The high-order bits of a continuation byte are always `10` . The bit representation of the code point is then divided across the low-order bits of the leading and continuation bytes.

Example (paraphrased from Wikipedia UTF-8 page (https://en.wikipedia.org/wiki/UTF-8))

Consider encoding the Euro sign, €.

a. The Unicode code point for € is U+20AC.
b. The code point is within the range U+0800 to U+FFFF and will require a three-byte sequence. There are 14 significant bits in the binary representation of 0x20AC.
c. Hex code point 20AC is binary `00100000 10101100` . Two leading zero bits of padding are used to fill to 16 bits. These 16 bits will be divided across the three-byte sequence.
d. Leading byte. High-order bits are fixed: three 1s followed by a 0 indicate the three-byte sequence. The low-order bits store the 4 highest bits of the code point. This byte is `11100010` . The code point has 12 bits remaining to be encoded.
e. Continuation byte. High-order bits are fixed `10` , low-order bits store the next 6 bits of the code point. This byte is `10000010` .
f. Continuation byte. High-order bits are fixed `10` , low-order bits store the last 6 bits of the code point. This byte is `10101100` .

The final three-byte sequence `11100010 10000010 10101100` can be more concisely written in hexadecimal, as `e2 82 ac` . The underscores indicate where the bits of the code point were distributed across the encoded bytes.

You are to write the `to_utf8` function that takes a Unicode code point and constructs its sequence of UTF-8 encoded bytes.

```
int to_utf8(unsigned short code_point, unsigned char seq[])
```

The `to_uft8` function has two parameters; an unsigned short `code_point` and a byte array `seq`. The function constructs the UTF-8 representation for the given code point and writes the sequence of encoded bytes into the array `seq`. The `seq` array is provided by the client and is guaranteed to be large enough to hold a full 3 bytes, although only 1 or 2 may be needed. The function returns the number of bytes written to the array (either 1, 2, or 3).

This task is excellent practice with constructing bitmasks and applying bitwise ops and standard techniques to extract/rearrange/pack bits. As always, custom sanitycheck (/class/cs107/sanitycheck.html) is your friend for thoroughly testing your work.

Advice/FAQ

The assignment writeup (this page) is the place where we spell out the formalities (required specifications, due date, grading standards, logistics, and so on). We also maintain a separate companion advice page for the assignment that offers informal recommendations and hints, along with answers to common student questions. Please check it out!

Go to advice/FAQ page (advice.html)

Grading

Below is the tentative grading rubric. We use a combination of automated tests and human review to evaluate your submission. More details are given in our page explaining How assignments are graded (/class/cs107/advice/assigngrade.html).

Readme questions (35 points)

- **readme.txt**. (35 points) For the code reading questions, you will be graded on the understanding of the issues demonstrated by your answers and the correctness of your conclusions. The requirement for the readme file is not intended to be onerous. Strunk and White famously say "vigorous writing is concise" and we couldn't agree more. These questions are intended be answered in just a sentence or two. Conserve your formal proof-writing efforts for CS103!

Code functionality (65 points)

- **Sanity cases**. (25 points) The default sanity check tests are used for sanity grading.
- **Comprehensive cases**. (38 points) We will thoroughly test on a variety of an additional inputs ranging from simple to complex, including edge conditions where appropriate.
- **Clean compile**. (2 points) We expect your code to compile cleanly without warnings.

Code quality (buckets weighted to contribute ~10 points)

The grader's code review is scored into buckets to emphasize the qualitative features of the review over the quantitative.

- *Bitwise manipulation*. We expect you to show proficiency in bitwise manipulation through clean construction and use of masks and proper application of the bitwise operations.
- *Algorithms.* Your chosen approach should demonstrate that you understand how to leverage bit patterns and numeric representation to directly and efficiently accomplish the task. Unnecessary divergence and special-case code should be avoided; unify into one general-purpose path wherever possible.
- *Style and readability.* We expect your code to be clean and readable. We will look for descriptive names, defined constants (not magic numbers!), and consistent layout. Be sure to use the most clear and direct C syntax and constructs available to you.
- *Documentation.* You are to document both the code you wrote and what we provided. We expect program overview and per-function comments that explain the overall design along with sparing use of inline comments to draw attention to noteworthy details or shed light on a dense or obscure passage. The audience for the comments is your C-savvy peer.

On-time bonus (+5%)

How CS107 handles deadlines and late work may differ from other classes, be sure you have read our complete late policy (/class/cs107/latepolicy.html).

The on-time bonus for this assignment is 5%. Submissions received by the due date earn the on-time bonus. The bonus is calculated as a percentage of the point score earned by the submission.

Finish and submit

Review the How to Submit (/class/cs107/submit.html) page for instructions. You can submit as many times as desired; your most recent submission is the one we will grade. Submitting a stable but unpolished/unfinished is like an insurance policy. If the unexpected happens and you miss the deadline to submit your final version, this previous submit will earn points. Without a submit, we cannot grade your work.

Submissions received by the due date receive the on-time bonus. If you miss the due date, late work may be submitted during the grace period without penalty. No submissions will be accepted after the grace period ends, please plan accordingly!

How did it go for you? Review the post-task self-check (/class/cs107/selfcheck.html#assign1).

Congratulations are in order on completing your first step on your journey in leveling up your systems knowledge and skills! Way to go and looking forward to more great things to come!

> And what exactly is a fascicle? It took me several hops through my dictionary (img/fasciculate.pdf) to find out!