Assignment 3: A Heap of Trouble

Due: Mon Oct 23 11:59 pm
Ontime bonus 5%. Grace period for late submissions until Wed Oct 25 11:59 pm

Assignment by Julie Zelenski and Michael Chang

Learning goals

This assignment should sharpen your skills in

- wrangling pointers & arrays
- managing memory allocation, both stack and heap
- using the C library I/O functions
- debugging memory errors (because we are sure you will encounter at least one or two :-)

Overview

Watch video walkthrough! (https://youtu.be/mhuZDYzckqM)

This assignment consists of two code-study exercises and two small programs to write. One code-study exercise concerns the `gets` function that reads a line from standard input, the other examines the `calloc` and `realloc` functions that allocate memory in the heap. The programs to write are simplified versions of the unix utilities `uniq` and `tail`.

Get started

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

```
git clone /afs/ir/class/cs107/repos/assign3/$USER assign3
```

The starter project contains `code.c` with the code for the code-study exercises, C files `mycat.c`, `read_line.c`, `mytail.c` and `myuniq.c`, and the supporting `Makefile`, `custom_tests`, and `readme.txt` files. In the `samples` subdirectory, you'll find our sample solutions and some sample input files.

1. Code study: gets

The standard C library function `gets` has an inherently awful design. It is intended to read a single line of text from stdin, stopping at the first newline character, and writing the read characters into the client's buffer. The fatal flaw of `gets` is that its only argument is the starting address of the buffer, with no indication of that buffer's length. Without the length, `gets` cannot tell when it should stop writing characters to avoid overflowing the buffer. There is absolutely no way to use `gets` safely. Its use has long been deprecated in favor of the properly-constrained `fgets` function.

For reasons of backward compatibility, `gets` lives on in the standard library and much effort goes into trying to dissuade programmers from using it. Let's look at how the tools try to impede use of the function and observe the consequences if we proceed past the warnings and use the function anyway. Answer the following questions in your readme.txt file:

a. The gcc compiler squawks about `gets` at compile-time. Use `make` to build the program and you'll see a special case complaint when anyone dares call `gets`. You should get not one but two warnings, one from the compiler and another from the linker. What are these warnings? (Fun fact: even if you invoke gcc with the -w flag to disable all warnings, the complaint from the linker lives on. You just can't shut that thing up!)
b. The MacOS libc (shown below as `apple_gets`) adds a shout-out against `gets` at runtime. Describe the behavior/output of `apple_gets` given its use of the `static` variable `warned`. (*Note:* it is highly irregular for the runtime behavior of a library function to produce unsolicited output like this. I guess someone really wanted to ensure that the use of `gets` didn't pass unnoticed!)
c. The `gets` function is implemented in musl (shown below as `musl_gets`) as a simple pass-through to the better-designed `fgets` function. Does this choice result in a `gets` function that actually operates more safely? Justify why or why not.
d. Review the use of the myth's version of `gets` in code.c. Run the code program and when prompted for your name, enter a response which will cause a buffer overflow. How many characters beyond the end do you have to go before you see any ill effect? How is the problem reported when it surfaces?

See the BUGS section of man gets for further harsh words against the function and hints of the security problems therein. Peter van der Linden's book *Expert C Programming: Deep C Secrets* summarizes the most famous gets exploit in the early bird gets() the Internet Worm (http://proquest.safaribooksonline.com.stanford.idm.oclc.org/book/programming/c/0131774298/2dot-it-not-a-bug-it-a-language-feature/ch02lev1sec3?#ch02lev2sec14). When we delve into the stack mechanics later this quarter, we will learn how a buffer overflow leads to these security problems. (To access the book content on Safari Books, you will need to authenticate with your SUNet id and may have to wait if there are too many concurrent users. The entire Expert C book is available and chock full of fascinating information -- I highly recommend it for its illuminating and comprehensive coverage of all things C!)

```
char *apple_gets(char *buf)
{
    int c;
    static int warned;
    static const char w[] = "warning: this program uses gets(), which is unsaf
e.\n";

    if (!warned) {
        fprintf(stderr, "%s", w);
        warned = 1;
    }
    char *s = buf;
    while ((c = getc(stdin)) != '\n') {
        if (c == EOF) {
            if (s == buf)
                return NULL;
            else
                break;
        } else {
            *s++ = c;
        }
    }
    *s = '\0';
    return buf;
}
```

```
char *musl_gets(char *s)
{
     char *ret = fgets(s, INT_MAX, stdin);
     if (ret && s[strlen(s)-1] == '\n') s[strlen(s)-1] = '\0';
     return ret;
}
```

2. Code study: calloc and realloc

Our programs will start making use of the heap as a client this week. Although we're not yet ready to dig into the internals of how a heap allocator operates, we can take a look at the implementation of some of the ancillary routines that coordinate with malloc and free .

Introduce yourself to malloc 's cousin calloc by reading its man page (man calloc) and then review the implementation for calloc shown below (taken from musl):

```
1    void *calloc(size_t m, size_t n)
2    {
3        if (n && m > SIZE_MAX/n) {
4            return NULL;
5        }
6        size_t sz = n * m;
7        void *p = malloc(sz);
8        if (p != NULL) {
9            size_t *wp;
10           size_t nw = (sz + sizeof(*wp) - 1)/sizeof(*wp);
11           for (wp = p; nw != 0; nw--, wp++)
12               if (*wp) *wp = 0;
13       }
14       return p;
15    }
```

The `calloc` interface is presented in terms of allocating an array of `m` elements each of size `n` bytes. This is somewhat misleading because there is nothing array-specific to its operation. It simply allocates a region of total size `m * n` bytes. The call `calloc(2, 5)` has the same effect as `calloc(5, 2)` or `calloc(1, 10)`. After allocating the space, it then zeros the memory, which is the feature that distinguishes `calloc` from ordinary `malloc`.

Answer the following questions in your readme.txt file:

a. Your colleague objects to the use of division on Line 3 because division is more expensive than multiplication and requires an extra check for a zero divisor. He proposes changing Line 3 to `if (n * m > SIZE_MAX)` claiming it will operate equivalently and more efficiently. Explain to him why his plan won't work.
b. Examine the expression on line 10. It bears a resemblance to a piece of code we studied before (lab1 (/class/cs107/lab1/)). What is being calculated by this expression? There is an assumption baked into this code about how many bytes `malloc` will actually reserve for a given requested size. What is that assumption? (This behavior is not a documented public feature and no client that was a true outsider should assume this, but `calloc` and `malloc` were authored by the same programmer, thus `calloc` is being written by someone with firsthand knowledge of the `malloc` internals and acting as a privileged insider.)
c. Your colleague is also displeased with Line 12. He claims that removing the `if (*wp)` and leaving just the `*wp = 0` would make the loop run faster with no change in what it accomplishes. Do you agree? Explain your reasoning.

The code shown below illustrates the behavior of a standard `realloc` function. Read the man page for `realloc` and take note of the behavior for unusual calls such realloc'ing a NULL pointer or resizing to zero bytes. Look for that special-case handling in the code below:

```
void *realloc(void *oldptr, size_t newsz)
{
    // this call is not a real function, used here as placeholder
    // for internal information managed by the heap allocator
    size_t oldsz = available_size_at_address(oldptr);

    if (newsz == 0) {
        free(oldptr);
        return NULL;
    }
    if (oldsz >= newsz) {
        return oldptr;
    }
    void *newptr = malloc(newsz);
    if (newptr != NULL && oldptr != NULL) {
        memcpy(newptr, oldptr, oldsz);
        free(oldptr);
    }
    return newptr;
}
```

Answer the following questions in your readme.txt file:

d. Whenever possible, the heap allocator prefers to keep the memory in-place and accommodate the new size at the existing location. What is the advantage of this option over the alternative of allocating a new piece of memory?

e. Why must the caller catch the return value from `realloc`? Why doesn't `realloc` just re-assign the pointer when it's necessary to move the memory block? (There is both a simple logistical reason and an underlying rationale for the design, either answer is acceptable.)
f. It is stated that a correct program should have a one-to-one correspondence between `malloc` calls and `free` calls. How does adding a `realloc` call change the needed number of `malloc` and `free` calls? Briefly explain your reasoning.

3. Write read_line

The C file-reading functions share a similar interface in which the client is responsible for providing the buffer to store the text read. The client must pre-allocate memory based on an expectation of what size will be "large enough". If the client's choice of size is too small, a line-reading function might overflow the buffer (`gets`) or truncate the line (`fgets`). A more client-friendly design would be to bundle the necessary allocation inside the read function and enlarge the space as necessary. Consider this function prototype for `read_line` function that operates in that manner:

```
char *read_line(FILE *fp);
```

The `read_line` function reads the next line from a file. The return value is a dynamically-allocated and null-terminated string. If the file contains no more lines to read, `read_line` returns NULL. This function performs like a snazzier version of `fgets` (in fact, internally it will be implemented using `fgets`) in that it reads the next line but this version also manages the allocation details and provides a cleaner interface for the client to use.

Specific details of the function's operation:

- The function reads the next line from the file. A line is considered to end after the first newline character or EOF, whichever comes first.
- If the line ended with a newline, the newline is not included in returned string, instead the newline is replaced with a null char. If the line to be read consists of just a newline character, the empty string would be returned.
- To allocate memory for the returned string, the function makes an initial allocation and if needed, successively enlarges it to accommodate the entire line. To be more specific, it first mallocs a buffer of minimum size (32) and reads the first part of the line into the buffer. If the line is longer than the original estimate, it reallocs the buffer to double its current size (64), and attempts to read the rest of the line. If there is still more to read, it reallocs to double the size again and so on until finally it reads to the newline or EOF that marks the end of the line.
- If unable to allocate sufficient memory, `read_line` should raise a fatal assert. As a habit, you should `assert` the result from every allocation request. Allocation failures are rare but deadly.
- The return value of `read_line` is the address of a dynamically-allocated piece of memory. The client is responsible for deallocating this memory with `free` when no longer needed.

You are to write the function `read_line` that operates in this fashion. Write your implementation in the `read_line.c` file. You will write and test this function in isolation now, and then will use the function later when writing the `mytail` and `myuniq` programs. You can test your `read_line` function using our provided `mycat.c` program. The `mycat` program is integrated with sanitycheck.

Review and comment starter code

Both `myuniq.c` and `mytail.c` are given to you with a small amount of code to handle the command-line arguments. Before starting on either program, first read and understand the given code, work out how to change/extend it to suit your needs, and finally add comments to document your strategy.

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

- The programs are set up to either read from stdin or a named file (using code similar to that studied in lab3 (/class/cs107/lab3/)). Will any special-case handling be required to handle stdin differently than a named file?
- `mytail` supports a command-line argument of `-number` to control the number of lines printed. How does the given code process that optional argument?
- What range of values is accepted as an argument for `mytail -number`?
- Do you see anything unexpected or erroneous? *We intend for our code to be bug-free; if you find otherwise, please let us know!*

As per usual, the code we provide has been stripped of its comments and it will be your job to provide the missing documentation.

4. Implement `myuniq`

What does the `uniq` command do?

Many standard unix commands operate as *filters* where they read input from a file, transform it in some way, and then print out the result. The unix utility `uniq` (`man uniq`) is one such filter. It removes adjacent matching lines from the input; the output will contain only one copy of each line that is duplicated in the input. When `uniq` is invoked with the `-c` flag, it also counts the number of consecutive occurrences for each line in the input. Consider the following sample use of `uniq -c` :

```
myth> cat samples/colors
red
green
green
red
blue
blue
blue
red
myth> uniq -c samples/colors
    1 red
    2 green
    1 red
    3 blue
    1 red
```

It's important to note that `uniq` does not detect repeated lines unless they are adjacent in the input.

How does `myuniq` operate?

The `myuniq` program you are to write is similar in operation to the standard `uniq` with these differences:

- `myuniq` prefixes each line of output with its count of consecutive occurrences in the input (this is the behavior of standard `uniq` when invoked with the `-c` flag)
- `myuniq` supports no command-line flags (standard `uniq` has a number of flags)

The implementation of `myuniq` is rather straightforward once you have a working `read_line` , the tricky part is being careful with memory allocation and deallocation. This program makes a nice warmup!

5. Implement `mytail`

What does the `tail` command do?

The unix `tail` command (`man tail`) is an example of another filter. This filter prints the final N lines of the input, where N is 10 (default value when not explicitly specified) or a number expressed as a command-line flag, e.g. `tail -4` . Consider the following sample use of `tail` :

```
myth> cat samples/colors
red
green
green
red
blue
blue
blue
red
myth> tail -3 samples/colors
blue
blue
red
```

If the input contains fewer than N total lines, all lines from the input are printed.

How does `mytail` operate?

The `mytail` program you are to write is similar in operation to the standard `tail` with these differences:

- `mytail` supports only one command-line flag, `-N` where N is the number of lines to output (standard `tail` has a number of other flags)
- `mytail` only reads one file either the named file (if specified) or standard input (if not) (standard `tail` can read from any number of file arguments)

While `mytail` is processing its input, it will need to keep track of N lines. For this you should use an array of char * pointers. However, N can potentially be a very large number, large enough that this array might exceed the capability of the stack. The starter code declares a stack array of a conservative maximum length. If N is <= this maximum, `mytail` should use this stack array. If N is larger, than `mytail` should instead malloc an array of pointers and use that instead.

At any given point, this array holds the most recent N lines read. The array is used to hold a "sliding window" of N lines that is moving its way through the input. When `mytail` hits the end of the input, the window will contain the final N lines.

When `mytail` starts processing the input, it will read the first N lines into the array. Upon reading the N+1st line, it should not enlarge the array, but instead should discard the 1st line of the input and use that array slot for the N+1st. The N+2nd overwrites the second line and so on. In this way, an array is used as a circular queue (https://en.wikipedia.org/wiki/Circular_buffer).

There is some logic to work out in the circular queue, but most of the challenge in this program concerns proper memory handling. By the time you are done with this assignment, you will be a heap master!

Valgrind

This assignment provides a lot of targeted practice in managing memory. You'll be using both the stack and heap for various purposes and need to take care to appropriately allocate and properly deallocate each. Bring your A game for this and be sure to get Valgrind on the job. It is critical that your code contain no memory errors. Plugging any pesky memory leaks is a more minor concern. You should also take care to monitor your memory efficiency and keep it on target. How can you do this?

Valgrind summarizes the overall heap memory usage. Look for a line at the end of a Valgrind report in this format:

```
total heap usage: 10 allocs, 10 frees, 888 bytes allocated
```

This is the aggregate total of all heap allocations during the program's lifetime. The count of allocs should always equal the count of frees. Fewer allocs than free is a sign of an error (free'd something that was not alloc'ed in first place), and fewer frees than allocs indicates you've got a leak. The count of allocations and total size allocated is a measure of your program's memory "footprint". Run our solution under Valgrind to see its footprint and compare yours to it as a measure of your program's memory efficiency. We very tightly specify the use of stack/heap memory for these programs, so the memory usage of a correct program should track the solution very closely. In grading, we usually give your programs some slack (say within 2-3x the sample) but you should aim for closer to validate your usage of memory is correct.

Advice/FAQ

Don't miss out on the good stuff in our companion document!

Go to advice/FAQ page (advice.html)

Grading

Here is the tentative point breakdown:

Readme questions (25 points)

- **readme.txt**. (25 points) For the code-study questions, you will be graded on the understanding of the issues demonstrated by your answers and the correctness of your conclusions.

Functionality (75 points)

- **Sanity cases** (25 points) Correct results on the default sanity check tests.
- **Comprehensive/stress cases** (30 points) Correct results for additional test cases with broad, comprehensive coverage and larger, more complex inputs.
- **Clean compile** (2 points) Compiles cleanly with no warnings.

- **Clean runs under valgrind** (15 points) Clean memory reports for all programs when run under valgrind. Memory errors (invalid read/write, use of freed memory, etc) are significant deductions. Memory leaks are a minor deduction. Every normal execution path is expected to run cleanly with no memory errors nor leaks reported. We will not test exceptional/error cases under Valgrind.
- **Reasonable efficiency** (4 points) We expect programs to be reasonably efficient in use of time and space. Full points are awarded for being on par (2-3x) with the sample program; deductions will apply for immoderate use of memory/time. There is no bonus for outperforming the benchmark (and extreme efforts to do so could detract from your code quality...).

Code review (buckets together weighted to contribute ~20 points)

- *Use of pointers and memory*. We expect you to show proficiency in handling pointers/memory, as demonstrated by appropriate use of stack versus heap allocation, no unnecessary levels of indirection, correct use of pointee types and typecasts, const correctness, and so on.
- *Program design*. We expect your code to show thoughtful design and appropriate decomposition. Data should be logically structured and accessed. Control flow should be clear and direct. When you need the same code in more than one place, you should unify, not copy and paste. If the C library provides functionality needed for a task, you should leverage these library functions rather than re-implement that functionality.
- *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%)

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. 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#assign3).