CS 107 Mid Quarter Assessment Practice

Andrew Benson

Disclaimer: These problems do not necessarily emulate the format of MQA problems. For example, the length and difficulty of the problems may differ from the actual MQA (if anything, longer and more difficult). There are also more practice problems here than you should expect on the MQA.

Problem 1: Entomology Tooling
(Topic: CS 107 Debugging / Tooling)

a. It's 11pm PT on the second late day of CS 107 assign4 and you've just finished writing 120 lines of C code without compiling it (not your best idea). Miraculously, there are no compilation errors when you run `make`. Unfortunately, when you run it on your first test case, all that's printed is `Segmentation fault (core dumped)`. Although it's disappointing, you're still determined to move forward and you want to find the line of code where the segmentation fault occurred as quickly as possible. What tool could you use to find this, and how would you use it?

Do a backtrace within `gdb`.
1) `gdb PROGRAM_NAME`
2) `run ARGUMENTS_TO_PROGRAM` (or `r ARGUMENTS_TO_PROGRAM`)
3) `backtrace` (or `bt`)

It might be reasonable to try `valgrind`, but it's not a direct way to find the line of code where the segfault occurred.

b. Following your answer to part a, you obtain this stack trace:

```
#0  __strcmp_sse2_unaligned () at ../sysdeps/x86_64/multiarch/strcmp-sse2-
unaligned.S:31
#1  0x0000000000400aef in cmp_pstr (p=0x7fffffffccf0, q=0x603240) at mysort.c:34
#2  0x000000000040109c in binsert (key=0x7fffffffccf0, base=0x603240,
p_nelem=0x7fffffffccc8, width=8,
    compar=0x400ab6 <cmp_pstr>) at util.c:30
#3  0x0000000000400cc0 in sort_lines (fp=0x603010, cmp=0x400ab6 <cmp_pstr>,
uniq=true, reverse=false) at mysort.c:90
#4  0x0000000000401018 in main (argc=3, argv=0x7fffffffde48) at mysort.c:149
```

Determine the file, function, and line number (in your code) where the segfault most immediately occurred.

`mysort.c`, in `cmp_pstr`, on line 34. Note that `__strcmp_sse2_unaligned` is not a function in your code – this is part of the implementation of `strcmp` for this system.

c. After studying that line of code, you don't think the root cause of the issue occurs there – you suspect that there's an error in some other part of your code that caused memory corruption, but it didn't cause the program to segfault until the above line. What tool could you use to find the first instance of memory corruption in your program?

`valgrind`

d. Later on, you're investigating a comparison function (one specifically for strings) in your code with the following function declaration:

```
int cmp_pstr(const void *p, const void *q);
```

You've used `gdb` to step into this function. You want to print out the string that `p` refers to. However, when you try to do so, all you get is this:

```
(gdb) print p
$1 = (const void *) 0x7fffffffccd0
```

What's wrong, and how should you change your gdb command so that it prints the string as desired? (Hint: remember that this is a comparison function. What are the arguments in a comparison function?)

Since this is a comparison function, `p` is a pointer to the string you want to print out. The print command only knows that `p` is a `void*`, so it just prints out the value of that pointer. Instead, try `print *(char**)p`.

Problem 2: After reading these problem titles you won't like puns one bit (Topic: Integer Representations)

Fill out the following table. Please use the 0b prefix when writing out binary representations and the 0x prefix when writing out hexadecimal representations. Write out leading zeroes for the given type.

Expression	Decimal Value	Binary Representation	Hex Representation
(signed char)(1 << 7)	`-128`	`0b10000000`	`0x80`
(unsigned char)(1 << 7)	`128`	`0b10000000`	`0x80`
(signed char)(0xF7F)	`127`	`0b01111111`	`0x7F`
(unsigned char)(0xF7F)	`127`	`0b01111111`	`0x7F`
(char)((~1) & (3 | 14))	`14`	`0b00001110`	`0x0E`

Problem 3: I bet Marx has a lecture on the types of classes (Topic: Integer Representations)

You've been hired by Stanford to optimize the memory consumption of their class scheduling system! Your first task is to use a C type to represent which days of the week that a class has lectures. Since there are only 7 days in a week, you decide to use a bitvector to encode this information, where the LSB represents whether there's lecture on Sunday, the next bit over Monday, and so forth. (Remember bitvectors from lecture 3? We used them to represent the set of CS courses taken by a student.) For example, here's how CS 107, which has lectures on Monday and Friday, would look:

0	1	0	0	0	1	0
Saturday lecture?	Friday lecture?	Thursday lecture?	Wednesday lecture?	Tuesday lecture?	Monday lecture?	Sunday lecture?

a. Recall that bitvectors are integer types. You want to pick the smallest integer type that has enough bits to represent the above information. Since you're trying to optimize for memory consumption, what type should you pick? Fill out the blank below.

```
unsigned _______char_______
```

b. In part a, we wrote `unsigned` for you. Does the sign of this type matter? Why or why not?

No. We are only interested in the bit representation of this value, not the value itself, so the sign does not matter.

c. You decide to call this type `week_schedule_t`. Write a `typedef` expression so that you're able to use `week_schedule_t` in your code in place of the type from part a.

```
typedef _______________unsigned char week_schedule_t_________________
```

d. Implement the following function, which takes in a `week_schedule_t` and a day of the week (0 = Sunday, 1 = Monday, etc) and returns a boolean representing whether there is lecture on that day for the given class.

```
bool has_lecture(week_schedule_t class, char day_of_week) {

    return ((class >> day_of_week) & 1) == 1;

}
```

e. Suppose you have two `week_schedule_t` values representing two of your classes. What is the real-life meaning of the following bitwise expressions? For example, the bitwise AND of them represents which days of the week you have lectures for both classes.

i. The bitwise OR (|) of the two values

which days of the week you have a lecture for either class

ii. The bitwise XOR (^) of the two values

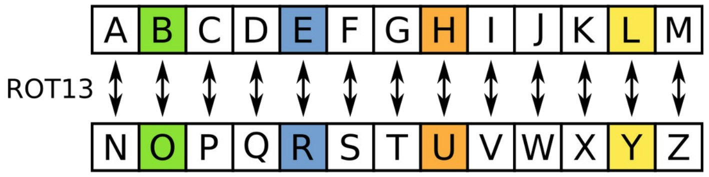
which days of the week you have a lecture for exactly one of the classes

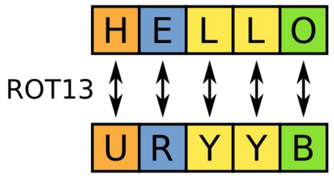
iii. The bitwise complement (~) of the one of the values

which days of the week you don't have a lecture for that class

Problem 4: Still a better encryption scheme than Facebook (Topic: Arrays, Structs, Strings)

ROT13 is a simple, trivially breakable substitution cipher for disguising text. It works by replacing each alphabetical character in the text with the character 13 over (thus, ROTate 13). Refer to the following diagram from Wikipedia:





https://en.wikipedia.org/wiki/ROT13

As in the diagram, "HELLO" becomes "URYYB" since 'H' is 13 characters away from 'U', 'E' is 13 characters away from 'R', and so forth.

You've been given an array of structs where the structs look like this:

```
struct rot13_data {
    char source[10];
    char rotated[10];
};
```

Each `struct rot13_data` contains room for two strings of **at most** 10 bytes (so the string could be shorter, even 1 character or empty!). Each string represents a single word composed of only lowercase English letters – so you shouldn't worry about having to deal with other symbols, like spaces. The `source` field already contains a word, but it's your job to rot13 it to fill out the `rotated` field – it currently contains uninitialized data!

a. Implement the following function, which fills out the `rotated` field with the rot13'ed version of the `source` field for every struct in the given array.

(You may assume without checking that all characters are lowercase English characters.)

```
void rot13_array_fill(struct rot13_data *array, size_t length) {
     for (unsigned int i = 0; i < length; i++) {
          size_t str_length = strlen(array[i].source);
          for (unsigned int j = 0; j < str_length; j++) {
               char source_letter = array[i].source[j];
               if (source_letter <= ‘m’) {
                    array[i].rotated[j] = source_letter + 13;
               } else {
                    array[i].rotated[j] = source_letter - 13;
               }
          }
          array[i].rotated[str_length] = ‘\0’;









     }
}
```

b. Write code to call this function to determine the rot13 value of "hello". Yes, you'll have to make an array containing only one element. Do not allocate any memory on the heap. You may use functions from the C string library.

```
struct rot13_data array[1];
```

```
strcpy(array[0].source, “hello”);
rot13_array_fill(array, 1);
```

Problem 5: Nearly Headless? How can your string be Nearly Headless? (Topic: Strings, Pointers)

Let’s define the nearly headless version of a string to be that string, except without the first character. The function `make_nearly_headless` takes in a string as an argument and makes it nearly headless by in-place removing the first character and returning nothing. You may assume the input string is not empty.

```
void make_nearly_headless(char *str);
```

Here’s some examples:

```
char s[] = “hello”;
make_nearly_headless(s);
printf(“%s\n”, s); // Prints “ello”
```

```
char s2[] = “nick”;
make_nearly_headless(s2);
printf(“%s\n”, s2); // Prints “ick”
```

a. Implement the function. You should not use any loops (prefer library functions).

```
void make_nearly_headless(char *str) {
     // Subtract 1 for first char, add 1 for NUL terminator
     size_t bytes_to_move = strlen(str);
     memmove(str, str + 1, bytes_to_move);




}
```

`memcpy` or `strcpy` would not work, as the memory overlaps!

b. Determine whether each of the following code snippets would compile and run correctly. If not, briefly explain why.

i.
```
char *s = “hello”;
make_nearly_headless(s);
```

No. The string is stored in read-only memory, so `make_nearly_headless` cannot modify it.

ii.
```
char s[] = “hello”;
make_nearly_headless(s);
```

Yes.

iii.
```
char s[] = {‘h’, ‘e’, ‘l’, ‘l’, ‘o’};
make_nearly_headless(s);
```

No. There is no NUL terminator in this string, so `strlen` will invoke undefined behavior.

iv.
```
char *s[] = “hello”;
make_nearly_headless(s);
```

No. This will not compile, as `char *s[]` expects an array of strings, not a single string.

v.
```
char *s[1];
s[0] = malloc(10 * sizeof(char));
strcpy(s[0], “hello”);
make_nearly_headless(s[0]);
```

Yes.

vi.
```
char *s = malloc(5 * sizeof(char));
strcpy(s, “hello”);
make_nearly_headless(s);
free(s);
```

No. There are only 5 bytes of memory in the block, so `strcpy` will write the NUL terminator into unallocated memory.

vii.
```
char *s = malloc(10 * sizeof(char));
strncpy(s, “hello”, strlen(“hello”));
make_nearly_headless(s);
free(s);
```

No. `strncpy` will only copy over 5 bytes, which does not include the NUL terminator.

Problem 6: why don't you just meet me in the middle? (Topic: Generics)

We are interested in the middle element of a generic array. Here, we'll define the middle as index `floor(num_elements / 2)`, so arrays with both an odd or even number of elements will have a middle element.

For example,

```
int array[] = {5, 4, 3, 2, 1};

int *ptr = get_middle(array, sizeof(int), 0); // "empty array"
assert(ptr == array); // Middle of empty is just the beginning.

int *ptr = get_middle(array, sizeof(int), 1); // "1-element array"
assert(ptr == array);
assert(*ptr == 5);

int *ptr = get_middle(array, sizeof(int), 4); // "4-element array"
assert(ptr == &array[2]);
assert(*ptr == 3);

int *ptr = get_middle(array, sizeof(int), 5); // 5-element array
assert(ptr == &array[2]);
assert(*ptr == 3);
```

However, remember that this function takes in a generic array – so the above should work analogously if we used a string array.

a. Implement `get_middle`, which returns a pointer to the middle element.

```
void *get_middle(void *array, size_t width, size_t num_elements) {
     size_t middle_index = num_elements / 2;
     return ((char*)array) + width * middle_index;




}
```

b. Why did `get_middle` return a pointer? Why couldn't it have returned the middle element itself?

Since the array is generic, the element could be any type with any size, so we wouldn't be able to write a return type for the function (as picking a return type makes it no longer generic). Instead, we return a pointer to the element so that the caller, who knows the actual type, can cast appropriately and dereference the pointer to get the element.

c. Suppose we want to make get_middle not return anything, and instead put the middle element into a location specified by the caller. Modify `get_middle` (including its parameters) to do this. Also show how you would modify the last example from above (the one that passed 5 for the number of elements) to assert that the middle element is 3 (don't heap-allocate memory).

```
void *get_middle(void *array, size_t width, size_t num_elements, void *out) {
     size_t middle_index = num_elements / 2;
     void *middle_ptr = ((char*)array) + width * middle_index;
     memcpy(out, middle_ptr, width);
}

int array[] = {5, 4, 3, 2, 1};
int x;
get_middle(array, sizeof(int), 5, &x);
assert(x == 3);
```

Problem 7: Reduce, Reuse, Recycle
(Topics: Generics, Function Pointers)

(Warning: this problem is overly complicated and takes a fair amount of time to understand.)

One common operation in functional programming (another style of programming that isn't studied much at Stanford   ) is called `foldl`. It has the following prototype:

```
void foldl(void *array, size_t width, size_t num_elems, void *base,
          void (*reducer)(void *a, void *b));
```

As you can see, it's...very generic. Its operation is best explained through **pseudocode** examples.

```
foldl([1, 2, 3, 4, 5], 0, add) => add(add(add(add(add(0, 1), 2), 3),
4), 5) => 15
```

```
foldl([“cat”, “and”, “dog”], “”, concat) => concat(concat(concat(“”,
“cat”), “and”), “dog”) => “catanddog”
```

`foldl` takes in an array, a “base element”, and a generic function, and applies the function pairwise starting with the base and first element, moving through the array left to right. It yields the final result.

It’s a bit harder to implement in a non-functional language like C, but it can be done with some modifications. The first three arguments should be familiar to you. The fourth is a pointer to a base element, and the fifth is a generic function that takes in two pointers to elements, combines them, and *puts the result in* `b`. So you can imagine applying reducer to the base and first array element and having the result overwrite the first array element, then applying reducer to the first array element and the second array element, overwriting the latter, and so forth, so that the final result ends up in the last element of the array. (As it turns out, we’re actually writing an in-place `scan` function (ish), but don’t worry about it.)

Here’s an example of calling `foldl` in C.

```
void in_place_sum(void *a, void *b) {
     int a_i = *(int*)a;
     int b_i = *(int*)b;
     *(int*)b = a_i + b_i;
}
```

```
int array[] = {1, 2, 3, 4, 5};
int base = 0;
foldl(array, sizeof(int), 5, &base, &in_place_sum);
// array now contains: {1, 3, 6, 10, 15}
```

Implement `foldl`.

```
void foldl(void *array, size_t width, size_t num_elems, void *base,
          void (*reducer)(void *a, void *b)) {
     for (unsigned int i = 0; i < num_elems; i++) {
          if (i == 0) {
               reducer(base, array);
          } else {
               void *ptr = (char*)array + i * width;
               reducer((char*)ptr - width, ptr);
          }
     }




}
```