CS 61C Fall 2022 — Memory and Floating Point — Discussion 3

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 Memory sectors are defined by the hardware, and cannot be altered.

False, The four major memory sectors, stack, heap, static/data, and text/code for any given process (application) are defined by the OS and may differ depending on what kind of memory is needed for it to run.

1.2 For large recursive functions, you should store your data on the heap over the stack.

False. If you store data on the heap in a recursive scheme, your malloc calls may lead to you rapidly running out of memory, or can lead to memory leaks as you lose where you allocate memory as each stack frame collapses.

1.3 True or False. The goals of floating point are to have a large range of values, a low amount of precision, and real arithmetic results

False, Floating point actually has HIGH precision.

1.4 True or False. The distance between floating point numbers increases as the absolute value of the numbers increase.

True

1.5 True or False. Floating Point addition is associative.

False. Because of rounding errors, (Small + Big) - Big != Small + (Big - Big)
FP approximates results because it only has 23 bits for Significand.

2 Memory Management

C does not automatically handle memory for you. In each program, an address space is set aside, separated in 2 dynamically changing regions and 2 'static' regions.

- The Stack: local variables inside of functions, where data is garbage immediately after the *function in which it was defined* returns. Each function call creates a stack frame with its own arguments and local variables. The stack dynamically changes, growing downwards as multiple functions are called within each other (LIFO structure), and collapsing upwards as functions finish execution and return. 先进后出
- The Heap: memory manually allocated by the programmer with `malloc`, `calloc`, or `realloc`. Used for data we want to persist beyond function calls, growing upwards to 'meet' the stack. Careful heap management is necessary to avoid Heisenbugs! Memory is freed only when the programmer explicitly frees it!
- Static data: global variables declared outside of functions, does not grow or shrink through function execution.

- Code (or Text): loaded at the start of the program and does not change after, contains executable instructions and any pre-processor macros.

There are a number of functions in C that can be used to dynamically allocate memory on the heap. The following are the ones we use in this class:

- `malloc(size_t size)` allocates a block of `size` bytes and returns the start of the block. The time it takes to search for a block is generally not dependent on `size`.
- `calloc(size_t count, size_t size)` allocates a block of `count * size` bytes, **sets every value in the block to zero**, then returns the start of the block.
- `realloc(void *ptr, size_t size)` "resizes" a previously-allocated block of memory to `size` bytes, returning the start of the resized block.
- `free(void *ptr)` deallocates a block of memory which starts at `ptr` that was previously allocated by the three previous functions.

2.1 For each part, choose one or more of the following memory segments where the data could be located: **code, static, heap, stack**.

(a) Static variables static

(b) Local variables stack

(c) Global variables code

(d) Constants code, static, stack

(e) Machine Instructions code

(f) Result of Dynamic Memory Allocation(`malloc` or `calloc`) heap

(g) String Literals static

2.2 Write the code necessary to allocate memory on the heap in the following scenarios

(a) An array `arr` of k integers

```
arr = (int*) malloc(k*sizeof(int));
```

(b) A string `str` containing p characters

```
str = (char*) malloc((p+1)*sizeof(char));
```

(c) An $n \times m$ matrix `mat` of integers initialized to zero.

```
mat = (int*) calloc(n*m, sizeof(int));
```

2.3 Compare the following two implementations of a function which duplicates a string. Is either one correct? Which one runs faster?

```
1  char* strdup1(char* s) {
2      int n = strlen(s);
3      char* new_str = malloc((n + 1) * sizeof(char));
4      for (int i = 0; i < n; i++) new_str[i] = s[i];
5      return new_str;
6  }
```

```
7  char* strdup2(char* s) {
8      int n = strlen(s);
9      char* new_str = calloc(n + 1, sizeof(char));
10     for (int i = 0; i < n; i++) new_str[i] = s[i];
11     return new_str;
12 }
```

The first implementation is incorrect because malloc doesn't initialize the allocated memory to any given value, so the new string may not be null-terminated. This is easily fixed, however, just by setting the last character in new_str to the null terminator. The second implementation is correct since calloc will set each character to zero so the string is always null-terminated.

Between the two implementations, the first will run slightly faster since malloc doesn't set the memory values, and thus run in O(1) time. calloc does set each memory location, so it runs in O(n) time. Effectively, we do "extra" work in the second implementation setting every character to zero, and then overwrite them with copied values afterwards.

2.4 What's the main issue with the code snippet seen here? (Hint: `gets()` is a function that reads in user input and stores it in the array given in the argument.)

```
1  char* foo() {
2      char buffer[64];
3      gets(buffer);
4
5      char* important_stuff = (char*) malloc(11 * sizeof(char));
6
7      int i;
8      for (i = 0; i < 10; i++) important_stuff[i] = buffer[i];
9      important_stuff[i] = '\0';
10     return important_stuff;
11 }
```

If the user input contains more than 63 characters, then the input will override other parts of the memory!

3 Linked List

Suppose we've defined a linked list **struct** as follows. Assume `*lst` points to the first element of the list, or is `NULL` if the list is empty.

```
struct ll_node {
    int first;
    struct ll_node* rest;
}
```

3.1 Implement `prepend`, which adds one new `value` to the front of the linked list. Hint: why use ll_node $**lst$ instead of ll_node$*lst$?

```
void prepend(struct ll_node** lst, int value) {
    struct ll_node** item = (struct ll_node*) malloc(sizeof(struct ll_node));
    item->first = value;
    item->rest = *lst;
    *lst = item;
}
```

3.2 Implement `free_ll`, which frees all the memory consumed by the linked list.

```
void free_ll(struct ll_node** lst){
    if (*lst){
        free_ll(&(*lst)->rest);
        free(*lst);
    }
    *lst = NULL;
}
```

4 Floating Point

The IEEE 754 standard defines a binary representation for floating point values using three fields.

- The *sign* determines the sign of the number (0 for positive, 1 for negative).
- The *exponent* is in **biased notation**. For instance, the bias is -127 which comes from $-(2^{8-1}-1)$ for single-precision floating point numbers.
- The *significand* or *mantissa* is akin to unsigned integers, but used to store a fraction instead of an integer.

The below table shows the bit breakdown for the single precision (32-bit) representation. The leftmost bit is the MSB and the rightmost bit is the LSB.

1	8	23
Sign	Exponent	Mantissa/Significand/Fraction

For normalized floats:

Value $= (-1)^{Sign} * 2^{Exp+Bias} * 1.\textbf{significand}_2$

For denormalized floats:

Value $= (-1)^{Sign} * 2^{Exp+Bias+1} * 0.\textbf{significand}_2$

Exponent	**Significand**	**Meaning**
0	Anything	Denorm
1-254	Anything	Normal
255	0	Infinity
255	Nonzero	NaN

Note that in the above table, our exponent has values from 0 to 255. When translating between binary and decimal floating point values, we must remember that there is a bias for the exponent.

4.1 Convert the following single-precision floating point numbers from binary to decimal or from decimal to binary. You may leave your answer as an expression.

- 0x00000000
- 8.25
- 0x00000F00
- 39.5625
- 0xFF94BEEF
- $-\infty$
- 1/3

5 More Floating Point Representation

As we saw above, not every number can be represented perfectly using floating point. For this question, we will only look at positive numbers.

5.1 What is the next smallest number larger than 2 that can be represented completely?

$(1+2^{-23})*2 = 2+2^{-22}$

5.2 What is the next smallest number larger than 4 that can be represented completely?

$(1+2^{-23})*4 = 4+2^{-21}$

5.3 What is the largest odd number that we can represent? Hint: Try applying the step size technique covered in lecture.

To find the largest odd number we can represent, we want to find when odd numbers will stop appearing. This will be with step size of 2.
As a result, plugging into Part 4: $2 = 2^{x-150} \rightarrow x = 151$
This means the number before $2^{151-127}$ was a distance of 1 (it is the first value whose stepsize is 2) and no number after will be odd. Thus, the odd number is simply subtracting the previous step size of 1. This gives, $2^{24}-1$.