CS 61C — C Part 2, Floating Point

Summer 2022 — Discussion 3

1 Pre-Check

1.1 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

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

1.3 True or False. Floating Point addition is associative.

2 Floating in the 61Sea

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 $= (-1)^{Sign} * 2^{Exp+Bias} * 1.\mathbf{significand}_2$
For denormalized floats:
Value $= (-1)^{Sign} * 2^{Exp+Bias+1} * 0.\mathbf{significand}_2$

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.

2.1 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

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

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

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

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

3 Pass-by-who?

3.1 Implement the following functions so that they work as described.

(a) Swap the value of two **int**s. *Remain swapped after returning from this function.*
Hint: Our answer is around three lines long.

```
void swap(________________, ________________) {
```

(b) Return the number of bytes in a string. *Do not use* `strlen`*.*
Hint: Our answer is around 4 lines long.

```
int mystrlen(________________) {
```

4 Debugging

4.1 The following functions may contain logic or syntax errors. Find and correct them.

(a) Returns the sum of all the elements in `summands`.

```
int sum(int *summands) {
    int sum = 0;
    for (int i = 0; i < sizeof(summands); i++)
        sum += *(summands + i);
    return sum;
}
```

(b) Increments all of the letters in the `string` which is stored at the front of an array of arbitrary length, `n >= strlen(string)`. Does not modify any other parts of the array's memory.

```
void increment(char *string, int n) {
    for (int i = 0; i < n; i++)
        *(string + i)++;
}
```

(c) Copies the string `src` to `dst`.

```
void copy(char *src, char *dst) {
    while (*dst++ = *src++);
}
```

(d) Overwrites an input string `src` with "61C is awesome!" if there's room. Does nothing if there is not. Assume that `length` correctly represents the length of `src`.

```
void cs61c(char *src, size_t length) {
    char *srcptr, replaceptr;
    char replacement[16] = "61C is awesome!";
    srcptr = src;
    replaceptr = replacement;
    if (length >= 16) {
        for (int i = 0; i < 16; i++)
            *srcptr++ = *replaceptr++;
    }
}
```

5 Allocation

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

(a) An array `arr` of k integers

(b) A string `str` containing p characters

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

6 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;
}
```

6.1 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)
```

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

```
void free_ll(struct ll_node** lst)
```