CS 61C
Spring 2022

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 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. Although floating point DOES
- Provide support for a wide range of values. (Both very small and very large)
- Help programmers deal with errors in real arithmetic because floating point can represent $+\infty$, $-\infty$, NaN (Not a number)

Floating point actually has HIGH precision. Recall that precision is a count of the number of bits in a computer word used to represent a value. Floating point helps you keep as much precision as possible because we have so much freedom to interpret our bits as whatever negative powers of 2 are useful for specifying the number.

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

True. The uneven spacing is due to the exponent representation of floating point numbers. There are a fixed number of bits in the significand. In IEEE 32 bit storage there are 23 bits for the significand, which means the LSB is 2^{-22} times the MSB. If the exponent is zero (after allowing for the offset) the difference between two neighboring floats will be 2^{-22}. If the exponent is 8, the difference between two neighboring floats will be 2^{-14} because the mantissa is multiplied by 2^8. Limited precision makes binary floating-point numbers discontinuous; there are gaps between them.

1.3 True or False. Floating Point addition is associative.

False. Because of rounding errors, you can find Big and Small numbers such that:
(Small + Big) + Big != Small + (Big + Big)
FP approximates results because it only has 23 bits for Significand

2 Memory Management

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

Static

(d) Constants

Code, static, or stack

Constants can be compiled directly into the code. `x = x + 1` can compile with the number 1 stored directly in the machine instruction in the code. That instruction will always increment the value of the variable `x` by 1, so it can be stored directly in the machine instruction without reference to other memory. This can also occur with pre-processor macros.

```
#define y 5

int plus_y(int x) {
    x = x + y;
    return x;
}
```

Constants can also be found in the stack or static storage depending on if it's declared in a function or not.

```
const int x = 1;

int sum(int* arr) {
    int total = 0;
    ...
}
```

In this example, `x` is a variable whose value will be stored in the static storage, while `total` is a local variable whose value will be stored on the stack. Variables declared **`const`** are not allowed to change, but the usage of **`const`** can get more tricky when combined with pointers.

(e) Machine Instructions

Code

(f) Result of `malloc`

Heap

(g) String Literals

Static.

When declared in a function, string literals can only be stored in static (or data) memory. String literals are declared when a character pointer is assigned to a string declared within quotation marks, i.e. `char* s = "string"`. Strings declared this was are stored in the static memory segment. You'll often see 3 other ways in which to declare strings, one being very similar to string literals, as following: `char[7] s = "string"`. This is a string array will be stored in the stack and is mutable. Note that the compilier will arrange for the char array to be initialised from the literal and be mutable.

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(sizeof(int) * k);
```

(b) A string `str` containing p characters

`str = (char *) malloc(sizeof(char) * (p + 1));` Don't forget the null terminator!

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

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

Alternative solution. This might be needed if you wanted to efficiently permute the rows of the matrix.

```
mat = (int **) calloc(n, sizeof(int *));
for (int i = 0; i < n; i++)
    mat[i] = (int *) calloc(m, sizeof(int));
```

2.3 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.)

```
char* foo() {
    char buffer[64];
    gets(buffer);

    char* important_stuff = (char*) malloc(11 * sizeof(char));

    int i;
    for (i = 0; i < 10; i++) important_stuff[i] = buffer[i];
    important_stuff[i] = '\0';
    return important_stuff;
}
```

If the user input contains more than 63 characters, then the input will override other parts of the memory! (You will learn more about this and how it can be used to maliciously exploit programs in CS 161.)

Note that it's perfectly acceptable in C to create an array on the stack. It's often discouraged (mostly because people often forget the array was initialized on the stack and accidentally return a pointer to it), but it's not an issue itself.

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;
}
```

2.4 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;
}
```

2.5 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; // Make writes to **lst fail instead of writing to unusable memory.
}
```

3 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.

3.1 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

  0

- 8.25

  0x41040000

- 0x00000F00

  $(2^{-12} + 2^{-13} + 2^{-14} + 2^{-15}) * 2^{-126}$

- 39.5625

  0x421E4000

- 0xFF94BEEF

  NaN

- -∞

  0xFF800000

- 1/3

  N/A — Impossible to actually represent, we can only approximate it

4 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.

$\boxed{4.1}$ What is the next smallest number larger than 2 that can be represented completely?

For this question, you increment the number by the smallest amount possible. This is the same as incrementing the significand by 1 at the rightmost location.
$(1 + 2^{-23}) * 2 = 2 + 2^{-22}$

$\boxed{4.2}$ What is the next smallest number larger than 4 that can be represented completely?

For this question, you increment the number by the smallest amount possible. This is the same as incrementing the significand by 1 at the rightmost location.
$(1 + 2^{-23}) * 4 = 4 + 2^{-21}$

$\boxed{4.3}$ 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,
$2^{24} - 1$