CS 61C Spring 2024 — Memory and Floating Point

Discussion 2

1 Pre-Check: Memory in C

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 If you try to dereference a variable that is not a pointer, what will happen? What about when you free one?

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

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

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.
- 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 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.

(d) Unallocating all but the first 5 values in an integer array `arr`. (Assume `arr` has more than 5 values)

2.2 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 }
```

3 Pre-Check: Floating Point

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:

3.1 The idea of floating point is to use the ability to move the radix (decimal) point wherever to represent a large range of real numbers as exact as possible.

3.2 Floating Point and Two's Complement can represent the same total amount of numbers (any reals, integer, etc.) given the same number of bits.

3.3 The distance between floating point numbers increases as the absolute value of the numbers increase.

3.4 Floating Point addition is associative.

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

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.

4.1 Convert the following single-precision floating point numbers from hexadecimal to decimal or from decimal to hexadecimal. 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?

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

5.3 What is the largest odd number that we can represent? Hint: At what power can we only represent even numbers?