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?

It will treat that variable's underlying bits as if they were a pointer and attempt to access the data there. C will allow you to do almost anything you want, though if you attempt to access an "illegal" memory address, it will segfault for reasons we will learn later in the course. It's why C is not considered "memory safe": you can shoot yourself in the foot if you're not careful. If you free a variable that either has been freed before or was not malloced/calloced/realloced, bad things happen. The behavior is undefined and terminates execution, resulting in an "invalid free" error.

1.2 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 operating system and may differ depending on what kind of memory is needed for it to run.

What's an example of a process that might need significant stack space, but very little text, static, and heap space? (Almost any basic deep recursive scheme, since you're making many new function calls on top of each other without closing the previous ones, and thus, stack frames.)

What's an example of a text and static heavy process? (Perhaps a process that is incredibly complicated but has efficient stack usage and does not dynamically allocate memory.)

What's an example of a heap-heavy process? (Maybe if you're using a lot of dynamic memory that the user attempts to access.)

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

False. Generally speaking, if you need to keep access to data over several separate function calls, use the heap. However, recursive functions call themselves, creating multiple stack frames and using each of their return values. 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.

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

```
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));
```

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

```
arr = realloc(arr, 5 * sizeof(int));
```

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

```
char* strdup1(char* s) {
    int n = strlen(s);
    char* new_str = malloc((n + 1) * sizeof(char));
    for (int i = 0; i < n; i++) new_str[i] = s[i];
    return new_str;
}
char* strdup2(char* s) {
    int n = strlen(s);
    char* new_str = calloc(n + 1, sizeof(char));
    for (int i = 0; i < n; i++) new_str[i] = s[i];
    return new_str;
}
```

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 need to set the memory values. calloc does set each memory location, so it runs in O(n) time in the worst case. Effectively, we do "extra" work in the second implementation setting every character to zero, and then overwrite them with the copied values afterwards.

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.

True. Floating point:
- Provides support for a wide range of values. (Both very small and very large)
- Helps programmers deal with errors in real arithmetic because floating point can represent + ∞, -∞, NaN (Not a number)
- Keeps high precision. Recall that precision is a count of the number of bits in a computer word used to represent a value. IEEE 754 allocates a majority of bits for the significand, allowing for the use of a combination of negative powers of two to represent fractions.

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.

False. Floating Point can represent infinities as well as NaNs, so the total amount of representable numbers is lower than Two's Complement, where every bit combination maps to a unique integer value.

3.3 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.

3.4 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.

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

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 hexadecimal to decimal or from decimal to hexadecimal. 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

We'll go more into depth with converting 8.25 and 0x00000F00. For the sake of brevity, the rest of the conversions can be done using the same process.

To convert 8.25 into binary, we first split up our 32b hexadecimal number into three parts. The sign is positive, so our sign bit -1^S will be 0. Then, we can solve for our significand. We know that our number will have a non-zero exponent, so we will have a leading 1 for our mantissa. Splitting 8.25 into its integer and decimal portions, we can determine that 8 will be encoded in binary as 1000. and 0.25 will be .01 (the 1 corresponds to the 2^{-2} place), so by implying the MSB, our significand will be 00001000.. Finally, we can solve for the exponent. As our leading 1 is in the 2^3 place to encode 8, we must use the bias in reverse to find what exponent we encode in binary. 130 added with a bias of -127 results in 3, so our exponent is 0b10000010. Our final binary number concatenated is 0 100 0001 0 000 0100 0000 0000 0000 0000, or 0x41040000.

For 0x00000F00, splitting up the hexadecimal gives us a sign bit and exponent bit of 0, and a significand of 0b 000 0000 0000 1111 0000 0000. We now know that this will be some sort of denormalized positive number. We can find out the true exponent by adding the bias + 1 to get the actual exponent of -126. Then, we can evaluate the mantissa by inspecting the bits that are 1 to the right of the radix point, and finding the corresponding negative power of two. This results in the mantissa evaluated as $2^{-12} + 2^{-13} + 2^{-14} + 2^{-15}$. Combining these get the extremely small number $(-1)^0 * 2^{-126} * (2^{-12} + 2^{-13} + 2^{-14} + 2^{-15})$

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?

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

5.2 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.
$(1 + 2^{-23}) * 4 = 4 + 2^{-21}$

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

To find the largest odd number we can represent, we want to find when odd numbers will stop appearing. This will be when the LSB will have a step size of 2, subtracted by 1. After this number, only even numbers can be represented in floating point.

We can think of each binary digit in the significant as corresponding to a different power of 2 to get to a final sum. For example, 0b1011 can be evaluated as $2^3 + 2^1 + 2^0$, where the MSB is the 3rd bit and corresponds to 2^3 and the LSB is the 0th bit at 2^0.

We want our LSB to correspond to 2^1, so by counting the number of mantissa bits (23) and including the implicit 1, we get a total exponent of 24. The smallest number with this power would have a mantissa of 00..00, so after taking in account the implicit 1 and subtracting, this gives $2^{24} - 1$