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 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.)

1.2 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.

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. 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 + ∞, -∞, 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.

1.4 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.

1.5 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

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 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;
    ...
```

```
6 }
```

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 Dynamic Memory Allocation(`malloc` or `calloc`)

Heap

(g) String Literals

Static.

When declared in a function, string literals can only be stored in static memory. String literals are declared when a character pointer is assigned to a string declared within quotation marks, i.e. **`char`**`* s = "string"`. You'll often see a near identical alternative to declaring a string: **`char`** `s[7] = "string"`. This string array will be stored in the stack (when declared inside a function) and is mutable, though they cannot change in size. Note that the compiler 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.

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

```
    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 set the memory values, and thus runs 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 the 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.)

```
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.

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$*lst$?

```
1  void prepend(struct ll_node** lst, int value) {
2      struct ll_node* item = (struct ll_node*) malloc(sizeof(struct ll_node));
3      item->first = value;
4      item->rest = *lst;
5      *lst = item;
6  }
```

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

```
1  void free_ll(struct ll_node** lst) {
2      if (*lst) {
3          free_ll(&((*lst)->rest));
4          free(*lst);
5      }
6      *lst = NULL; // Make writes to **lst fail instead of writing to unusable memory.
7  }
```

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

  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

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: 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,
$2^{24} - 1$