CS 61C Spring 2023

C and Memory

Discussion 2

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 if false, correct the statement to make it true:

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

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

2 Pass-by-who?

2.1 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 5 lines long.

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

3 Bit-wise Operations

3.1 In C, we have a few bit-wise operators at our disposal:

- `AND` (`&`)
- `NOT` ($\sim$)
- `OR` (`|`)
- `XOR` ($\wedge$)
- `SHIFT LEFT` (`<<`)
  - Example: `0b0001 << 2 = 0b0100`
- `SHIFT RIGHT` (`>>`)
  - Example: `0b0100 >> 2 = 0b0001`

a	**b**	**a & b**	**a | b**	**a $\wedge$ b**	**$\sim$a**
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

For your convenience, truth tables for the logical operators are provided above. With the binary numbers `a`, `b`, and `c` below, perform the following bit-wise operations:

```
a = 0b1000 1011
b = 0b0011 0101
c = 0b1111 0000
```

(a) `a & b`

(b) `b` $\wedge \sim$ `c`

(c) `a | 0`

(d) `a | (c >> 5)`

3.2 Bitwise operators are frequently useful for implementing functions on specific bits of a value. For the following operations, describe what they do to the input `x` (an arbitrary 8-bit integer) :

(a) `x | 0b10000001`

(b) `x & 0b00000001`

(c) `x` $\wedge$ `0b11110000`

4 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.

4.1 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

(b) Local variables

(c) Global variables

(d) Constants (constant variables or values)

(e) Functions (i.e. Machine Instructions)

(f) Result of Dynamic Memory Allocation(`malloc` or `calloc`)

(g) String Literals

4.2 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.

4.3 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];
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 }
```

4.4 What's the main issue with the code snippet seen here? (`gets()` is a function that reads in user input and stores it in the array given in the argument.)

```
1  char* foo() {
2      char buffer[64];
3      gets(buffer);
4
5      char* important_stuff = (char*) malloc(11 * sizeof(char));
6
7      int i;
8      for (i = 0; i < 10; i++) important_stuff[i] = buffer[i];
9      important_stuff[i] = '\0';
10     return important_stuff;
11 }
```

5 Endianness

Compare these different ways of storing 'DEADBEEF'. Assume that each program is run on the same machine and architecture.

```
char[] arr = 'DEADBEEF'
```

```
int arr[2];
arr[0] = 0xDEADBEEF;
arr[1] = 0x00000000;    //null terminator in hex is 0x00
```

5.1 Do these two C programs store 'DEADBEEF' in memory the same way?

You take a look at the ASCII table and translate the string 'DEADBEEF' into bytes.

```
int arr[2];
arr[0] = 0x44454144
//stores 'DEAD' in ascending order in arr[0]
arr[1] = 0x42454546
//stores 'BEEF' in ascending order in arr[1]
```

5.2 Does this C program store 'DEADBEEF' in memory the same way as storing it as a string?