CS 61C Fall 2024 — C Discussion 2

1 Pre-Check: 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 True or False: C is a pass-by-value language.

1.2 In compiled languages, the compile time is generally pretty fast, however the run-time is significantly slower than interpreted languages.

1.3 What is a pointer? What does it have in common to an array variable?

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

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

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

(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

2.2 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.

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

2.3 Compare the following two implementations of a function which duplicates a string. Is either one correct?

```
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;
}
```

3 C Generics

3.1 **True or False:** In C, it is possible to directly dereference a `void *` pointer, e.g.

```
... = *ptr;
```

3.2 Generic functions (i.e., generics) in C use void * pointers to operate on memory without the restriction of types. Such generics pointers do not support dereferencing, as the number of bytes to access from memory is not known at compile-time. They instead use byte handling functions such as `memcpy` and `memmove`.

Implement `rotate`, which will prompt the following program to generate the provided output.

```
int main(int argc, char *argv[]) {
  int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  print_int_array(array, 10);
  rotate(array, array + 5, array + 10);
  print_int_array(array, 10);
  rotate(array, array + 1, array + 10);
  print_int_array(array, 10);
  rotate(array + 4, array + 5, array + 6);
  print_int_array(array, 10);
  return 0;
}
```

Output:

```
$ ./rotate
Array: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Array: 6, 7, 8, 9, 10, 1, 2, 3, 4, 5
Array: 7, 8, 9, 10, 1, 2, 3, 4, 5, 6
Array: 7, 8, 9, 10, 2, 1, 3, 4, 5, 6
```

Your Solution:

```
void rotate(void *front, void *separator, void *end) {








}
```

4 Pass-by-who?

4.1 Consider the following blocks of C code:

```
void printall(int *x) {
    // Suppose x points to 0xABDE2464
    const int NUM_ELEMS = 3;
    for(int i = 0; i < NUM_ELEMS; i += 1) {
        printf("Address: %x \n", x);
        x++;
    }
}
```

(a) What three memory addresses are printed by this program?

```
void printall(char *x) {
    // Suppose x points to 0xABDE2464
    const int NUM_ELEMS = 3;
    for(int i = 0; i < NUM_ELEMS; i += 1) {
        printf("Address: %x \n", x);
        x++;
    }
}
```

(b) What three memory addresses are printed by this program?

4.2 The following functions may contain logic or syntax errors. Find and correct them.

(a) Returns the sum of all the elements in `summands`.

```
int sum(int *summands) {
    int sum = 0;
    for (int i = 0; i < sizeof(summands); i++)
        sum += *(summands + i);
    return sum;
}
```

(b) Increments all of the letters in the `string` which is stored at the front of an array of arbitrary length, `n <= strlen(string)`. Does not modify any other parts of the array's memory.

```
void increment(char *string, int n) {
    for (int i = 0; i < n; i++)
        *(string + i)++;
}
```

4.3 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(________________) {
```

5 Endianness

- Machines are byte-addressable. Memory is like a large array of cells. Each storage cell stores 8 bits, and these byte cells are ordered with an address.
- A 32b architecture has 32 bit memory addresses, addresses 0x00000000 - 0xFFFFFFFF

Typed variables

- Examples: int, long, char
- sizeof(dataType) indicates the number of bytes in memory required to store a particular data type

Pointers

- a variable whose value is an address of another variable
- Declaration: dataType* name;
- Dereference operator: Based on the pointer declaration statement, the compiler fetches the corresponding amount of bytes. For example, if p is a pointer to a 4 byte integer variable x, then *p involves fetching 4 bytes starting from the address of x, which is the value of p. Therefore, the value of x and value of *p are equal

Endianness

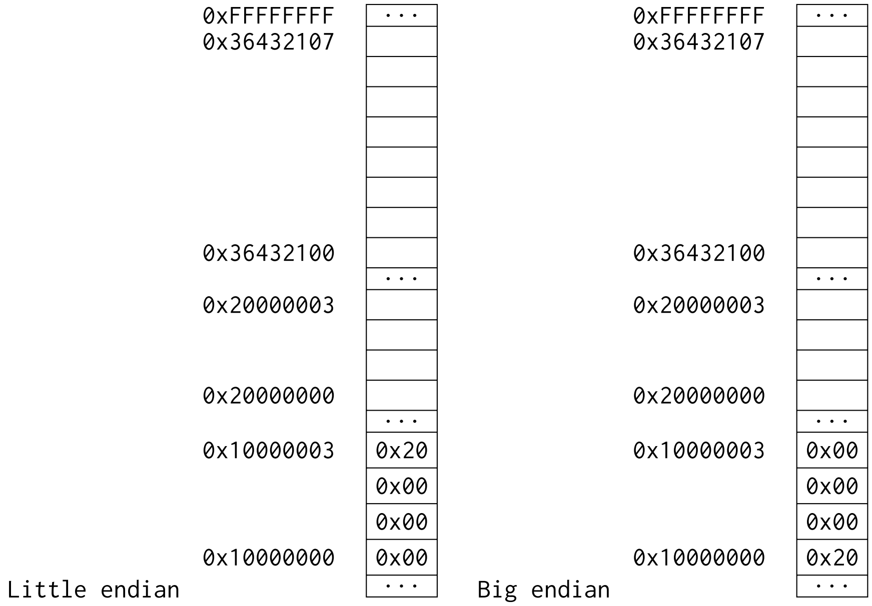
- Recall different data types are stored in x amount of contiguous byte cells in memory
- Big endian: the most significant byte of the value of a variable is stored in memory at the lowest address of the chunk of byte cells allocated for that variable
- Little endian: the least significant byte of the value of a variable is stored in memory at the lowest address of the chunk of byte cells allocated for the variable

5.1 Based on the following code and a 32b architecture, fill in the values located in memory at the byte cells for both a big endian and little endian system.

Suppose:

- the array nums starts at address 0x36432100
- p's address is 0x10000000

```
uint32_t nums[2] = {10, 20};
uint32_t* q = (uint32_t*) nums;
uint32_t** p = &q;
```



5.2 Provide two answers for the following questions: big endian system and little endian system

Suppose `uint64_t`* y = (`uint64_t`*) nums is executed after the code

1. What does *y evaluate to?

2. What does `&q` evaluate to? What does `&nums` evaluate to?

3. What does *(q+1) evaluate to?