CS 61C Summer 2024 — C — 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 = malloc(sizeof(int) * k);
```

(b) A string `str` containing p characters

`str = malloc(sizeof(char) * (p + 1));` Don't forget the null terminator!

(c) An $n \times m$ matrix `mat` of integers initialized to zero.

`mat = calloc(n * m, sizeof(int));` Alternative solution. This might be needed if you wanted to efficiently permute the rows of the matrix.

```c
mat = calloc(n, sizeof(int *));
for (int i = 0; i < n; i++)
    mat[i] = calloc(m, sizeof(int));
```

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

```c
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?

```c
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 Review: Introduction to 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:

3.1 The correct way of declaring a character array is `char[] array`.

False. The correct way is `char array[]`.

3.2 True or False: C is a pass-by-value language.

True. If you want to pass a reference to anything, you should use a pointer.

4 Pass-by-who?

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

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

It is necessary to pass a size alongside the pointer.

```c
int sum(int* summands, size_t n) {
    int sum = 0;
    for (int i = 0; i < n; 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.

The ends of strings are denoted by the null terminator rather than n. Simply having space for n characters in the array does not mean the string stored inside is also of length n.

```c
void increment(char* string) {
    for (i = 0; string[i] != 0; i++)
        string[i]++; // or (*(string + i))++;
}
```

Another common bug to watch out for is the corner case that occurs when incrementing the character with the value 0xFF. Adding 1 to 0xFF will overflow back to 0, producing a null terminator and unintentionally shortening the string.

(c) Overwrites an input string `src` with "61C is awesome!" if there's room. Does nothing if there is not. Assume that `length` correctly represents the length of `src`.

```c
void cs61c(char *src, size_t length) {
    char *srcptr, replaceptr;
    char replacement[16] = "61C is awesome!";
    srcptr = src;
    replaceptr = replacement;
    if (length >= 16) {
        for (int i = 0; i < 16; i++)
            *srcptr++ = *replaceptr++;
    }
}
```

`char *srcptr, replaceptr` initializes a `char` pointer, and a `char`—not two `char` pointers.

The correct initialization should be, `char *srcptr, *replaceptr`.

4.2 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.

```c
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

(b) Return the number of bytes in a string. *Do not use* `strlen`. Hint: Our answer is around 5 lines long.

```c
int mystrlen(char* str) {
    int count = 0;
    while (*str != 0) {
        str++;
        count++;
    }
    return count;
}
```

5 C Generics

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

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

False. To dereference a pointer, we must know the number of bytes to access from memory at compile time. Generics employ generic pointers and therefore cannot use the dereference operator!

5.2 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.

```c
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:

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







}
```

```c
void rotate(         ,            ,           ) {
  size_t width = (char *) end - (char *) front;
  size_t prefix_width = (char *) separator - (char *) front;
  size_t suffix_width = width - prefix_width;
  char temp[prefix_width];
  memcpy(temp, front, prefix_width);
  memmove(front, separator, suffix_width);
  memcpy((char *) end - prefix_width, temp, prefix_width);
}
```