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?

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

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

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

(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)

2.2 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 }
```

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`.

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

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`.

```
1  int sum(int *summands) {
2      int sum = 0;
3      for (int i = 0; i < sizeof(summands); i++)
4          sum += *(summands + i);
5      return sum;
6  }
```

(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.

```
1  void increment(char *string, int n) {
2      for (int i = 0; i < n; i++)
3          *(string + i)++;
4  }
```

(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`.

```
1  void cs61c(char *src, size_t length) {
2      char *srcptr, replaceptr;
3      char replacement[16] = "61C is awesome!";
4      srcptr = src;
5      replaceptr = replacement;
6      if (length >= 16) {
7          for (int i = 0; i < 16; i++)
8              *srcptr++ = *replaceptr++;
```

4 *C*

```
9       }
10  }
```

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.

```
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 C Generics

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

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

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.

```
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) {



}
```