CS61C Spring 2025

C Memory Management

Discussion 2

1 Memory Management

1.1 For each part, choose one or more of the following memory segments where the data could be located: **code, static, heap, stack**

a) Local variables

b) Global variables

c) Constants (constant variables or values)

d) Functions (i.e. Machine Instructions)

e) Results of Dynamic Memory Allocation (`malloc` or `calloc`)

f) String Literals

1.2 Write the code necessary to allocate memory **on the heap** in the following scenarios:

(a) An array of `arr` of `k` integers

(b) A string `str` of length `p`. Note that a string's length is defined by `strlen`

(c) An `n` × `m` matrix `mat` of integers initialized to zero.

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

```
int *arr = ... ;
```

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

```
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 }
```

```
1 char* strdup2(char* s) {
2     int n = strlen(s);
3     char* new_str = calloc(n + 1, sizeof(char));
4     for (int i = 0; i < n; i++) new_str[i] = s[i];
5     return new_str;
6 }
```

2 Pass-by-Who?

2.1 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++;
9      }
10 }
```

2.2 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.

```
1  void swap(_________________, _________________) {




}
```

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

```
1  int mystrlen(_________________) {




}
```

3 Endianness

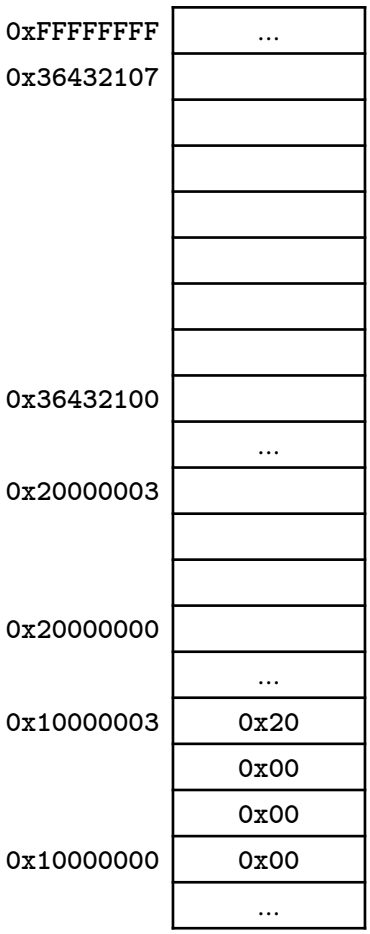
3.1 Suppose we run the following code on a 32b architecture:

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

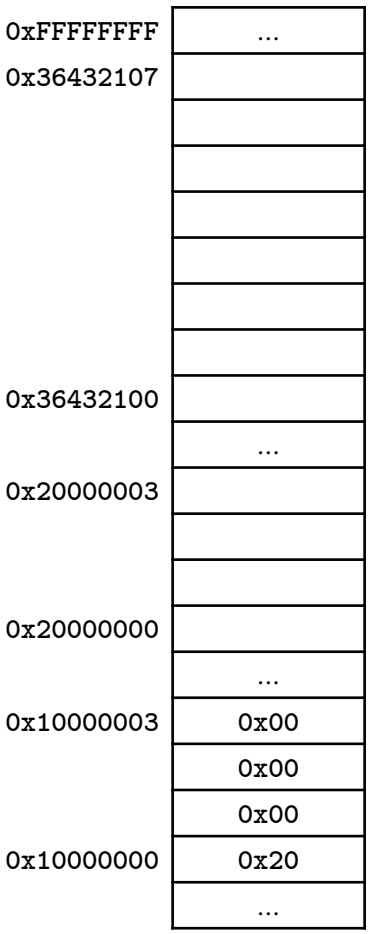
Find the values located in memory at the byte cells for both a Big Endian and a Little Endian system given that:

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

Little Endian



Big Endian



3.2 Suppose we add an an additional instruction (line #4) to the end of the previous code block:

```
1  uint32_t nums[2] = {10, 20};
2  uint32_t  *q = (uint32_t *) nums;
3  uint32_t **p = &q;
4  uint64_t  *y = (uint64_t *) nums;
```

Provide answers for the following questions for both a Big Endian system and Little Endian system:

1) What does `*y` evaluate to?

2) What does `&q` evaluate to?

3) What does `&nums` evaluate to?

4) What does `*(q + 1)` evaluate to?

4 C Generics

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

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

4.2 Generic functions (i.e., generics) in C use `void *` pointers to operate on memory without the restriction of types. Generic 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) {




}
```