1 C

C is syntactically similar to Java, but there are a few key differences:

1. C is function-oriented, not object-oriented; there are no objects.

2. C does not automatically handle memory for you.

   - Stack memory, or things allocated the way you’re accustomed to: data is garbage immediately after the function in which it was defined returns.

   - Heap memory, or things allocated with malloc, calloc, or realloc commands: data is freed only when the programmer explicitly frees it!

   - In any case, allocated memory always holds garbage until it is initialized!

3. C uses pointers explicitly. \*p tells us to use the value that p points to, rather than the value of p, and &x gives the address of x rather than the value of x.

On the left is the memory represented as a box-and-pointer diagram.

On the right, we see how the memory is really represented in the computer.

```
Let’s assume that int* p is located at 0xF9320904 and int x is located at 0xF93209B0. As we can observe:

   - *p should return 0x2A (4210).
   - p should return 0xF93209AC.
   - x should return 0x61C.
   - &x should return 0xF93209B0.

Let’s say we have an int **pp that is located at 0xF9320900.
```

1.1 What does pp evaluate to? How about *pp? What about **pp?
The following functions are syntactically-correct C, but written in an incomprehensible style. Describe the behavior of each function in plain English.

(a) Recall that the ternary operator evaluates the condition before the ? and returns the value before the colon (:) if true, or the value after it if false.

```c
int foo(int *arr, size_t n) {
    return n ? arr[0] + foo(arr + 1, n - 1) : 0;
}
```

(b) Recall that the negation operator, !, returns 0 if the value is non-zero, and 1 if the value is 0. The ~ operator performs a bitwise not (NOT) operation.

```c
int bar(int *arr, size_t n) {
    int sum = 0, i;
    for (i = n; i > 0; i--)
        sum += !arr[i - 1];
    return ~sum + 1;
}
```

(c) Recall that ^ is the bitwise exclusive-or (XOR) operator.

```c
void baz(int x, int y) {
    x = x ^ y;
    y = x ^ y;
    x = x ^ y;
}
```

## 2 Programming with Pointers

Implement the following functions so that they work as described.

(a) Swap the value of two ints. Remain swapped after returning from this function.

```c
void swap(
```

(b) Return the number of bytes in a string. Do not use strlen.

```c
int mystrlen(
```
The following functions may contain logic or syntax errors. Find and correct them.

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

```c
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, \( n \geq \text{strlen(string)} \). Does not modify any other parts of the array's memory.

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

(c) Copies the string `src` to `dst`.

```c
void copy(char* src, char* dst) {
    while (*dst++ = *src++);
}
```

(d) 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++;
    }
}
```
3 Memory Management

3.1 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
(e) Machine Instructions
(f) Result of malloc
(g) String Literals

3.2 Write the code necessary to allocate memory on the heap in the following scenarios

(a) An array \texttt{arr} of \(k\) integers
(b) A string \texttt{str} containing \(p\) characters
(c) An \(n \times m\) matrix \texttt{mat} of integers initialized to zero.

Suppose we’ve defined a linked list \texttt{struct} as follows. Assume \texttt{*lst} points to the first element of the list, or is \texttt{NULL} if the list is empty.

\begin{verbatim}
struct ll_node {
    int first;
    struct ll_node* rest;
}
\end{verbatim}

3.3 Implement \texttt{prepend}, which adds one new value to the front of the linked list.

\begin{verbatim}
void prepend(struct ll_node** lst, int value)
\end{verbatim}

3.4 Implement \texttt{free_ll}, which frees all the memory consumed by the linked list.

\begin{verbatim}
void free_ll(struct ll_node** lst)
\end{verbatim}