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 that are not manually allocated: data is garbage immediately after the function in which it was defined returns.
   - Heap memory, or things allocated with malloc, calloc, or realloc: data is freed only when the programmer explicitly frees it!
   - There are two other sections of memory that we learn about in this course, static and code, but we’ll get to those later.
   - In any case, allocated memory always holds garbage until it is initialized!

3. C uses pointers explicitly. If \( p \) is a pointer, then \( *p \) 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.

<table>
<thead>
<tr>
<th>0xFFFFFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>0xF93209B0</td>
</tr>
<tr>
<td>0xF93209AC</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>0xF9320904</td>
</tr>
<tr>
<td>0xF9320900</td>
</tr>
<tr>
<td>0x00000000</td>
</tr>
</tbody>
</table>

Let’s assume that \textbf{int*} \( p \) is located at 0xF9320904 and \textbf{int} \( x \) is located at 0xF93209B0. As we can observe:

- \( *p \) evaluates to 0x2A (42_{10}).
- \( p \) evaluates to 0xF93209AC.
- \( x \) evaluates to 0x61C.
- \( &x \) evaluates to 0xF93209B0.

Let’s say we have an \textbf{int **pp} that is located at 0xF9320900.
1.1 What does \texttt{pp} evaluate to? How about \texttt{*pp}? What about \texttt{**pp}?

1.2 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 \texttt{?} and returns the value before the colon (\texttt{:}) if true, or the value after it if false.

\begin{verbatim}
int foo(int *arr, size_t n) {
    return n ? arr[0] + foo(arr + 1, n - 1) : 0;
}
\end{verbatim}

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

\begin{verbatim}
int bar(int *arr, size_t n) {
    int sum = 0, i;
    for (i = n; i > 0; i--)
        sum += !arr[i - 1];
    return \textasciitilde sum + 1;
}
\end{verbatim}

(c) Recall that \texttt{\textasciicircum} is the bitwise exclusive-or (XOR) operator.

\begin{verbatim}
void baz(int x, int y) {
    x = x \textasciicircum y;
    y = x \textasciicircum y;
    x = x \textasciicircum y;
}
\end{verbatim}

(d) (Bonus: How do you write the bitwise exclusive-nor (XNOR) operator in C?)

2 Programming with Pointers

2.1 Implement the following functions so that they work as described.

(a) Swap the value of two \texttt{ints}. \textit{Remain swapped after returning from this function.}

\begin{verbatim}
void swap(
\end{verbatim}
(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
```
```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 `arr` of `k` integers
(b) A string `str` containing `p` characters
(c) An `n × m` matrix `mat` of integers initialized to zero.

### 3.3
What’s the main issue with the code snippet seen here? (Hint: `gets()` is a function that reads in user input and stores it in the array given in the argument.)

```c
char* foo() {
    char* buffer[64];
    gets(buffer);
    char* important_stuff = (char*) malloc(11 * sizeof(char));
    int i;
    for (i = 0; i < 10; i++) important_stuff[i] = buffer[i];
```
Suppose we’ve defined a linked list `struct` as follows. Assume `*lst` points to the first element of the list, or is NULL if the list is empty.

```c
struct ll_node {
    int first;
    struct ll_node* rest;
}
```

Implement `prepend`, which adds one new value to the front of the linked list. Hint: why use `ll_node** lst` instead of `ll_node* lst`?

```c
void prepend(struct ll_node** lst, int value)
```

Implement `free_ll`, which frees all the memory consumed by the linked list.

```c
void free_ll(struct ll_node** lst)
```