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.

```
0xFFFFFFF  ...  0xFFFFFFF  ...
0xF93209B0 x=0x61C  0xF93209B0  0x61C
0xF93209AC  0x2A   0xF93209AC  0x2A
  ...        ...       ...
0xF9320904  p    0xF9320904  0xF93209AC
0xF9320900  pp  0xF9320900  0xF9320904
  ...        ...       ...
0x00000000  ...  0x00000000  ...
```

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

- *p evaluates to 0x2A (42₁₀).
- p evaluates to 0xF93209AC.
- x evaluates to 0x61C.
- &x evaluates to 0xF93209B0.

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

\texttt{pp} evaluates to \texttt{0xF9320904}. \texttt{*pp} evaluates to \texttt{0xF93209AC}. \texttt{**pp} evaluates to \texttt{0x2A}.

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 ? and returns the value before the colon (:) 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}

Returns the sum of the first \textit{N} elements in \texttt{arr}.

(b) Recall that the negation operator, \texttt{!}, returns 0 if the value is non-zero, and 1 if the value is 0. The \texttt{\~} operator performs a \textit{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 \~sum + 1;
}
\end{verbatim}

Returns \texttt{-1} times the number of zeroes in the first \textit{N} elements of \texttt{arr}.

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

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

Ultimately does not change the value of either \texttt{x} or \texttt{y}.

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(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
\end{verbatim}

(b) Return the number of bytes in a string. \textit{Do not use strlen}.
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, \( n \geq \text{strlen(string)} \). 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) Copies the string `src` to `dst`.

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

No errors.

(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] = "C 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.

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
Static
(b) Local variables
Stack
(c) Global variables
Static
(d) Constants
Code, static, or stack

Constants can be compiled directly into the code. x = x + 1 can compile with the number 1 stored directly in the machine instruction in the code. That instruction will always increment the value of the variable x by 1, so it can be stored directly in the machine instruction without reference to other memory. This can also occur with pre-processor macros.

#define y 5
int plus_y(int x) {
    x = x + y;
    return x;
}

Constants can also be found in the stack or static storage depending on if it’s declared in a function or not.

const int x = 1;
int sum(int* arr) {

int total = 0;
...
}

In this example, x is a variable whose value will be stored in the static storage, while total is a local variable whose value will be stored on the stack. Variables declared const are not allowed to change, but the usage of const can get more tricky when combined with pointers.

(e) Machine Instructions

Code

(f) Result of malloc

Heap

(g) String Literals

Static or stack.

When declared in a function, string literals can be stored in different places. char* s = "string" is stored in the static memory segment while char[7] s = "string" will be stored in the stack.

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

(a) An array arr of k integers

    arr = (int *) malloc(sizeof(int) * k);

(b) A string str containing p characters

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

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

    mat = (int *) calloc(n * m, sizeof(int));

    Alternative solution. This might be needed if you wanted to efficiently permute the rows of the matrix.

    mat = (int **) calloc(n, sizeof(int *));

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

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.

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) {
    struct ll_node* item = (struct ll_node*) malloc(sizeof(struct ll_node));
    item->first = value;
    item->rest = *lst;
    *lst = item;
}
```

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

```c
void free_ll(struct ll_node** lst) {
    if (*lst) {
        free_ll(&(*lst)->rest);
        free(*lst);
    }
    *lst = NULL; // Make writes to **lst fail instead of writing to unusable memory.
}
```