1 Advanced C

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;
}
```

1.1 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;
}
```

1.2 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.
}
```

2 Memory Management

2.1 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.

```c
#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.

```c
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 \texttt{const} are not allowed to change, but the usage of \texttt{const} can get more tricky when combined with pointers.

(e) Machine Instructions

Code

(f) Result of \texttt{malloc}

Heap

(g) String Literals

Static or stack.

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

2.2 Write the code necessary to allocate memory on the heap in the following scenarios.
(a) An array \( \text{arr} \) of \( k \) integers

\[
\text{arr} = (\text{int} *) \text{malloc}(\text{sizeof(int)} * k);
\]

(b) A string \( \text{str} \) containing \( p \) characters

\[
\text{str} = (\text{char} *) \text{malloc}(\text{sizeof(char)} * (p + 1)); \text{Don’t forget the null terminator!}
\]

(c) An \( n \times m \) matrix \( \text{mat} \) of integers initialized to zero.

\[
\text{mat} = (\text{int} *) \text{calloc}(n * m, \text{sizeof(int)});
\]

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

\[
\begin{align*}
1 & \quad \text{mat} = (\text{int} **) \text{calloc}(n, \text{sizeof(int *)}); \\
2 & \quad \text{for (int } i = 0; i < n; i++) \\
3 & \quad \quad \text{mat}[i] = (\text{int} *) \text{calloc}(m, \text{sizeof(int)});
\end{align*}
\]

2.3 What is wrong with the C code below?

\[
\begin{align*}
1 & \quad \text{int* } \text{pi} = \text{malloc}(314 * \text{sizeof(int)}); \\
2 & \quad \text{if (!raspberry) } \\
3 & \quad \quad \text{pi} = \text{malloc}(1 * \text{sizeof(int)}); \\
4 & \quad \}
5 & \quad \text{return pi};
\end{align*}
\]

There’s a memory leak if \( \text{raspberry} \) is false as the original value of \( \text{pi} \) will be unreachable.