1 C Memory Management

1. Match the items on the left with the memory segment in which they are stored. Answers may be used more than once, and more than one answer may be required.

1. Static variables B
2. Local variables D
3. Global variables B
4. Constants A, B, D
5. Machine Instructions A
6. Result of malloc() C
7. String Literals B

<table>
<thead>
<tr>
<th>A. Code</th>
<th>B. Static</th>
<th>C. Heap</th>
<th>D. Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static variables</td>
<td>Local variables</td>
<td>Result of malloc</td>
<td>String Literals</td>
</tr>
</tbody>
</table>

4 Explained: With DEFINE, you can replace something in the code. With const declaration, it's stored in the static or stack depending on being declared in a function or not.

2. Write the code necessary to properly allocate memory (on the heap) in the following scenarios

1. An array arr of k integers
   
   ```
   arr = (int *) malloc(sizeof(int) * k);
   ```

2. A string str containing p characters
   
   ```
   str = (char *) malloc(sizeof(char) * (p + 1)); // Don't forget the null terminator!
   ```

3. An n x m matrix mat of integers initialized to zeros
   
   ```
   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));
   }
   ```

3. What is wrong with the C code below?

   ```
   int* pi = malloc(314 * sizeof(int));
   if(!raspberry) {
     pi = malloc(1 * sizeof(int)); // Memory leak if not raspberry
   }
   return pi;
   ```

4. Write code to prepend (add to the start) to a linked list, and to free/empty the entire list.

   ```
   struct ll_node { struct ll_node* next; int value; }
   ```
   ```
   void prepend(struct ll_node** lst, int val)
   ```
   ```
   struct ll_node* item = (struct ll_node*) malloc(sizeof(struct ll_node));
   item->value = val;
   item->next = *lst;
   *lst = item;
   ```
   ```
   void free_ll(struct ll_node** lst)
   ```
   ```
   if(*lst) {
     free_ll(&(*lst)->next);
     free(*lst);
   }
   /* so that attempts to write to **lst fail instead of writing to unusable memory */
   *lst = NULL;
   ```

   Note: *lst points to the first element of the list, or is NULL if the list is empty.
2 Data Structures in C

In this question, we will implement a array-based stack of integers in C. The stack will be represented by the struct below.

```c
struct stack {
    int size;  // Number of elements in stackArray
    int topIndex;  // Index in array of top of stack, or -1 if empty
    int *stackArray;  // Array holding elements of stack
}
```

Implement the functions below.

```c
// Create a new stack with the given array size
struct stack *init_stack(int size) {
    struct stack *stk = (struct stack *) malloc(sizeof(struct stack));
    if (!stk) {
        return NULL;
    }
    stk->size = size;
    stk->topIndex = -1;
    stk->stackArray = (int *) malloc(sizeof(int) * size);
    if (!(stk->stackArray)) {
        free(stk);
        return NULL;
    }
    return stk;
}
```

// Add the given element to the stack. Resize by doubling the array size if full.
// Return 1 on success, 0 on failure. stk should be unchanged on failure.
```c
int push(int x, struct stack* stk) {
    if (stk->topIndex == stk->size - 1) {
        int *stackArray = (int *) realloc(stk->stackArray, stk->size * 2)
        if (!stackArray) {
            return 0;
        }
        stk->stackArray = stackArray;
        stk->size = stk->size * 2;
    }
    stk->topIndex++;
    stk->stackArray[stk->topIndex] = x;
    return 1;
}
```

// Remove and return the top element from the stack. Return 0 if empty.
```c
int pop(struct stack *stk) {
    if (stk->topIndex < 0) {
        return 0;
    }
    int x = stk->stackArray[stk->topIndex];
    // Setting to 0 is not strictly needed, but can be useful for
    // debugging/assertions
    stk->stackArray[stk->topIndex] = 0;
    stk->topIndex--;
    return x;
}
```
3 RISC-V Intro

1. Assume we have an array in memory that contains `int* arr = {1,2,3,4,5,6,0}`. Let the value of `arr` be a multiple of 4 and stored in register `s0`. What do the snippets of RISC-V code do? Note that these snippets all run immediately after each other on the same core (snippet a runs, then snippet b and so on).

   a) `lw t0, 12(s0)`
   Sets register t0 equal to `arr[3]`

   b) `slli t1, t0, 2`  
   `add t2, s0, t1`  
   `lw t3, 0(t2)`  
   `addi t3, t3, 1`  
   `sw t3, 0(t2)`
   Increments the array element specified by t0 (i.e. `arr[t0]`) by 1

   c) `lw t0, 0(s0)`
   `xori t0, t0, 0xFFF ;Immediates are sign-extended (0xFFF -> all 1s)`;
   `addi t0, t0, 1`
   Sets the register t0 to the two’s complement negation of `arr[0]`

2. What are the instructions to branch to label on each of the following conditions? The only branch instructions you may use are `beq` and `bne`.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Instruction</th>
<th>Instruction</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s0 &lt; s1</code></td>
<td><code>slt t0, s0, s1</code></td>
<td><code>slt t0, s1, s0</code></td>
<td><code>sltiu t0, s0, 2</code></td>
</tr>
<tr>
<td></td>
<td><code>bne t0, 0, label</code></td>
<td><code>beq t0, 0, label</code></td>
<td><code>beq t0, 0, label</code></td>
</tr>
</tbody>
</table>
## 4 Translating between C and RISC-V

Translate between the C and RISC-V code. You may want to use the RISC-V Reference Card for more information on the instruction set and syntax. In all of the C examples, we show you how the different variables map to registers – you don’t have to worry about the stack or any memory-related issues. You may assume all registers are initialized to zero.

<table>
<thead>
<tr>
<th>C</th>
<th>RISC-V</th>
</tr>
</thead>
<tbody>
<tr>
<td>// s0 -&gt; a, s1 -&gt; b</td>
<td>addi s0, x0, 4</td>
</tr>
<tr>
<td>// s2 -&gt; c, s3 -&gt; z</td>
<td>addi s1, x0, 5</td>
</tr>
<tr>
<td>int a = 4, b = 5, c = 6, z;</td>
<td>addi s2, x0, 6</td>
</tr>
<tr>
<td>z = a + b + c + 10;</td>
<td>add s3, s0, s1</td>
</tr>
<tr>
<td></td>
<td>add s3, s3, s2</td>
</tr>
<tr>
<td></td>
<td>addi s3, s3, 10</td>
</tr>
<tr>
<td></td>
<td>sw x0, 0(s0)</td>
</tr>
<tr>
<td></td>
<td>addi s1, x0, 2</td>
</tr>
<tr>
<td></td>
<td>sw s1, 4(s0)</td>
</tr>
<tr>
<td></td>
<td>slli t0, s1, 2</td>
</tr>
<tr>
<td></td>
<td>add t0, t0, s0</td>
</tr>
<tr>
<td></td>
<td>sw s1, 0(t0)</td>
</tr>
<tr>
<td>// s0 -&gt; int * p = intArr;</td>
<td>addi s0, x0, 5</td>
</tr>
<tr>
<td>// s1 -&gt; a;</td>
<td>addi s1, x0, 10</td>
</tr>
<tr>
<td>*p = 0;</td>
<td>add t0, s0, s0</td>
</tr>
<tr>
<td>int a = 2;</td>
<td>bne t0, s1, else</td>
</tr>
<tr>
<td>p[1] = p[a] = a;</td>
<td>xor s0, x0, x0</td>
</tr>
<tr>
<td></td>
<td>jal x0, exit</td>
</tr>
<tr>
<td></td>
<td>else:</td>
</tr>
<tr>
<td></td>
<td>addi s1, s0, -1</td>
</tr>
<tr>
<td></td>
<td>exit:</td>
</tr>
<tr>
<td></td>
<td>addi s0, x0, 0</td>
</tr>
<tr>
<td></td>
<td>addi s1, x0, 1</td>
</tr>
<tr>
<td></td>
<td>add t0, x0, 30</td>
</tr>
<tr>
<td></td>
<td>loop:</td>
</tr>
<tr>
<td></td>
<td>beq s0, t0, exit</td>
</tr>
<tr>
<td></td>
<td>add s1, s1, s1</td>
</tr>
<tr>
<td></td>
<td>addi s0, s0, 1</td>
</tr>
<tr>
<td></td>
<td>jal x0, loop</td>
</tr>
<tr>
<td></td>
<td>exit:</td>
</tr>
<tr>
<td></td>
<td>addi s1, s1, 0</td>
</tr>
<tr>
<td></td>
<td>loop:</td>
</tr>
<tr>
<td></td>
<td>beq s0, x0, exit</td>
</tr>
<tr>
<td></td>
<td>add s1, s1, s0</td>
</tr>
<tr>
<td></td>
<td>add s0, s0, -1</td>
</tr>
<tr>
<td></td>
<td>jal x0, loop</td>
</tr>
<tr>
<td></td>
<td>exit:</td>
</tr>
<tr>
<td></td>
<td>addi s1, s1, 0</td>
</tr>
<tr>
<td></td>
<td>loop:</td>
</tr>
<tr>
<td></td>
<td>beq s0, x0, exit</td>
</tr>
<tr>
<td></td>
<td>add s1, s1, s0</td>
</tr>
<tr>
<td></td>
<td>add s0, s0, -1</td>
</tr>
<tr>
<td></td>
<td>jal x0, loop</td>
</tr>
<tr>
<td></td>
<td>exit:</td>
</tr>
</tbody>
</table>