Pointers

• All data is in memory. That means that each data has a memory address that maps to it. Pointers are variables that contain the address values.
• You can dereference a pointer (put a * in front of it) to obtain the data in the given address.
• When you initialize a pointer, you only make room in the memory for the space to hold the pointer, NOT the space to hold the data it’s pointing to!
• You can obtain the address of any data by putting & in front of the variable name.

Write functions that achieve the given tasks. Not all of them necessarily have a solution.

1. Swaps the value of two ints declared in main.

   ```c
   void swap(int* a, int* b){
       int temp = *a;
       *a = *b;
       *b = temp;
   }
   ```

2. Increments the value of an int declared in main by one.

   ```c
   void increment(int *x) {
       (*x)++;
   }
   /* OR */
   void increment(int *x) {
       x[0]++;
   }
   ```

3. Returns the number of bytes in the input string. Does not use strlen.

   ```c
   int mystrlen(char* str) {
       int count = 0;
       while (*str++) {
           count++;
       }
       return count;
   }
   ```

4. Returns the number of elements in the input array ARR of ints.

   You can’t. C has no way to determine an end of a sequence of ints.
Memory Management in C

5. In which memory sections (code, static, heap, stack) do the following reside?
   ```c
   #define val 16
   char arr[] = "foo";
   void foo(int arg){
       char *str = (char *) malloc(val);
       char *ptr = arr;
   }
   ```
   arg Stack  arr Static  str Stack  *str  Heap  val Code (used in instructions)

6. What are two reasons we might need to use malloc in a C program?
   - Persistence – Need to allocate memory that stays allocated beyond function exit, but which can be freed at will.
   - Dynamic allocation – Amount of memory to be allocated only known during runtime.
   - Error checking – Can check during runtime whether or not a particular allocation is available. (if there’s enough space)

7. What is wrong with the C code below?
   ```c
   int* ptr = malloc(4*sizeof(int));
   if(extra_large){
       ptr = malloc(10*sizeof(int));
   }
   return ptr;
   ```
   If extra_large is true, we have a memory leak (we lose the pointer to the memory initially allocated).

MIPS

- 32 Registers, $16$-$17 => $s0$-$s7$, $8$-$15 => $t0$-$t7$. $0$ is reserved for the value 0, and cannot be overwritten with other values.
  THERE ARE NO VARIABLES IN MIPS, JUST REGISTERS.
- MIPS Instruction Format: Operand Dest, Src1, Src2 (In most cases)
- Some example MIPS Instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add dest, src0, src1</td>
<td>add $s0$, $s1$, $s2$</td>
</tr>
<tr>
<td>addi</td>
<td>addi dest, src0, immediate</td>
<td>addi $s0$, $s1$, 12</td>
</tr>
<tr>
<td>sll / srl</td>
<td>sll dest, src, immediate</td>
<td>sll $t0$, 4($s0)</td>
</tr>
<tr>
<td>lw / lb</td>
<td>lw dest, offset(base addr)</td>
<td>lw $t0$, 4($s0)</td>
</tr>
<tr>
<td>sw / sb</td>
<td>sw src, offset(base addr)</td>
<td>sw $t0$, 4($s0)</td>
</tr>
</tbody>
</table>

C       | MIPS                                      |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>// $s0$ -&gt; a (use $s0$ for a),</td>
<td>addi $s0$, $0$, 4</td>
</tr>
<tr>
<td>// $s1$ -&gt; b</td>
<td>addi $s1$, $0$, 5</td>
</tr>
<tr>
<td>// $s2$ -&gt; c, $s3$ -&gt; z</td>
<td>addi $s2$, $0$, 6</td>
</tr>
<tr>
<td>int a=4, b=5, c=6, z;</td>
<td>add $s3$, $s0$, $s1$</td>
</tr>
<tr>
<td>z = a+b+c+10;</td>
<td>add $s3$, $s3$, $s2$</td>
</tr>
<tr>
<td>/ $s0$ -&gt; int *p = (int *)malloc</td>
<td>add $s3$, $s3$, 10</td>
</tr>
<tr>
<td>/ $s1$ -&gt; a</td>
<td>sw $0$, 0($s0)</td>
</tr>
<tr>
<td>p[0] = 0;</td>
<td>addiu $s1$, $0$, 2</td>
</tr>
<tr>
<td>int a = 2;</td>
<td>sw $s1$, 4($s0)</td>
</tr>
<tr>
<td>p[1] = a;</td>
<td>sll $t0$, $s1$, 2 #multiply by 4</td>
</tr>
<tr>
<td>p[a] = a;</td>
<td>addu $t1$, $t0$, $s0$</td>
</tr>
<tr>
<td></td>
<td>sw $s1$, 0($t1)$</td>
</tr>
</tbody>
</table>