C Coding Guerilla Section Solutions 1b (C coding)

1. **BuggC (Spring 2014 MT1 #4)**

A colleague of yours has implemented some homebrew C99 string manipulation functions, while steadfastly refusing to use any standard libraries, but they're buggy! We've marked each potentially problematic line with // <*number*>. Your job is to fill in a correct replacement line in the corresponding row of the following table, or write 'OK' if there is nothing wrong. DO NOT LEAVE ANY FIELDS BLANK.

Line number	Replacement Code
1	for(char c = *s; (c = *s) != '\0'; s++) {
2	*s += 'a' – 'A';
3	OK
4	OK
5	odds += numbers[i] & 1 OR odds += *(numbers+i) & 1

```
/** Converts the string S to lowercase */
void string_to_lowercase(char *s) {
    for(char c = *s; c != '\0'; s++) {   // 1
        if(c >= 'A' && c <= 'Z') {
            s += 'a' - 'A';              // 2
        }
    }
}
```

With the code as-is, the loop will never terminate since *c* never changes. Since we want *c* to always be the character we're converting to lowercase, we can update *c* every loop to the value that *s*, which we're incrementing, points at. On line 2, what we want to do is change the character pointed to by *s* to lowercase. However, the current code updates the pointer itself, and not the character that is pointed at. To fix this, we just dereference *s* to update the character being pointed at.

```
/** Returns the number of bytes in S before, but not counting, the null terminator. */
size_t string_length(char *s) {
    char *s2 = s;
    while(*s2++);          // 3
    return s2 - s - 1;     // 4
}
```

On line 3, the increment operator takes precedence over the dereference operator, so this while loop will increment the pointer *s2* while checking if *s2* points at a null character (this denotes the end of a string). When the while loop finishes, *s2* should point at the address 1 after the null character, so we must subtract 1 from *s2* - *s* to get the correct string length.

```
/** Return the number of odd numbers in a number array */
uint32_t number_odds(uint32_t *numbers, uint32_t size) {
    Uint32_t odds = 0;
    for (uint32_t i = 0; i < size; i++)
        odds += *numbers+i && 1;     // 5
    return odds;
}
```

The first thing that we can notice is that the wrong & operator is used on line 5. The && operator is used here, which is a logical AND for boolean conditions. What we want to use is the bitwise & operator, which will mask the number we're currently looking at and give us only the least significant bit. If this bit is 1, then we have an odd number, otherwise, we have an even number. The next thing that needs to be fixed is the pointer arithmetic for retrieving the correct number to check the oddness of. The code currently adds *i* after dereferencing the *numbers* pointer, which is incorrect. We want to retrieve the integer at the address *numbers + i*, so we can either put parentheses around that clause, or we can use the array retrieval operator (*numbers[i]*) to achieve this effect.

2. **Insert TreeNode**

Implement the following function. TreeNode struct is reprinted below for your reference.

```
struct TreeNode {
    int32_t value;
    struct TreeNode *left;
    struct TreeNode *right;
    struct TreeNode *parent;
    uint64_t magic; // you are not required to modify this
}
```

Insert a TreeNode with value, val, under parent. If the parent does not have a left child, make the TreeNode the left child. If the parent has a left child, then make the TreeNode the right child. It is guaranteed that there will always be a spot under parent for you to insert the new node. Note: If parent is NULL, you are creating the root. This function should return a pointer to the newly inserted TreeNode. Not all lines are needed.

```
struct TreeNode * insertNode(struct TreeNode *parent, int val) {
        // Allocating memory for TreeNode
        struct TreeNode *node =
                (struct TreeNode *) malloc(sizeof(struct TreeNode));
        node->value = val; // setting value of node to val
        node->left = NULL;
        node->right = NULL;
        node->parent = parent; // setting parent of node to parent
        if(parent) {
                if(!parent->left) {
                        parent->left = node;
                } else {
                        parent->right = node;
                }
        }
        return node;
}
```

You first want to allocate enough space in the heap for a TreeNode. You can then set the value of the node to the value specified as the function argument and set its left and right child to being NULL. Now you must take care of connecting the child node you created to the parent specified and vice versa. You can first go ahead and assign the specified parent as the node's parent. Before assigning the new node to the parent, first check that the parent is not null. Remember if the parent value is NULL, then the node you just created is the root node and has no parent node. Once that it is guaranteed, if the parent does not have the left child, assign the new node to the left child of the parent. If not, assign the new node to the right child of the parent.

3. **Number Pushing and Popping** (Fall 2016 M1 #3)

Your task is to implement a simple stack adding machine that uses a stack data structure. For example Push 2 and Push 3, PopAdd yields 5 in the top of the stack. Following this with Push 1, PopAdd would yield 6. Fill in the code for functions push and popadd so they meet the specifications stated in the comments. Do not make any other code modifications. Calls to malloc always returns a valid address.

```
/* Each item on the stack is represented
   by a pointer to the previous element
   (NULL if none) and its value. */
typedef struct stack_el {
     struct stack_el *prev;
     double val;
} stack_el;
```

```
/* PUSH: Push new value to top of the stack. Return
  pointer to new top of stack. */
stack_el push(stack_el *top_of_stack, double v) {
    stack_el* new_top = (stack_el*) malloc (sizeof(stack_el));
    new_top -> prev = top_of_stack;
    new_top -> val = v;
    return new_top;
}

/* POPADD: Pop top stack element and add its value
  To the new top's value. Return new top of stack.
  Free no longer used memory. Do not change
  the stack if it has fewer than 2 elements. */
stack_el* popadd(stack_el *top_of_stack) {
    if(top_of_stack == NULL) {
            return 0;
    }

    if(!(top_of_stack -> prev == NULL)) {          // check for <2 elems
            top_of_stack -> prev -> val += top_of_stack -> val;
            stack_el* new_top = top_of_stack -> prev;
            free(top_of_stack);
            return new_top;
    }
    return top_of_stack; // return top if <2 elements
}
```

4. A lot of Pointing

Assume you are given an int array arr, with a pointer p to its beginning:

int arr[] = {0x61c, 0x5008, 0xd, 0x4, 0x3, 0x4ffc};
int *p = arr;

Suppose arr is at location 0x5000 in memory, i.e., the value of p if interpreted as an integer is 0x5000. To visualize this scenario:

0x61c	0x5008	0xd	0x4	0x3	0x4ffc

arr[0] **…** **arr[5]**

Assume that integers and pointers are both 32 bits. What are the values of the following expressions? If an expression may cause an error, write "Error" instead.

a)*(p+3) = _________0x4________ d) *(int*)(p[1]) = ______0xd (13)_________

b) p[4] = ________0x3_________ e) *(int*)(*(p+5)) = ____error (out of bounds)___

c) *(p+5) + p[3] = __________0x5000__________

a) This is equivalent to p[3], which contains the value 0x4 (arrays are zero-indexed). Alternatively, you can think about it as adding three to pointer and dereferencing it. So p+3 now starts at 0x4 and goes on. Dereferencing p+3 will get the value stored at p+3, which is 0x4.

b) p[4] gets the fourth element in the array, which is 0x3 (arrays are zero-indexed).

c) From a), we know that *(p+5) is equivalent to p[5], so the expression in c) is equivalent to p[5] + p[3] which is equivalent to 0x4ffc + 0x4. 0xc + 0x4 is equivalent to 16, so in hexadecimal, this is a 0x10, which means carrying 1. 0xf + 0x1 is also 16, so it is equivalent to 0x10, which is a 1 carry. This chain continues until we reach the 4, which becomes a 5. Thus, the final value for the expression becomes 0x5000.

d) p[1] contains 0x5008, and this value is then casted to an integer array. This means that (int*)(p[1]) is equivalent to an array that starts at 0x5008. Dereferencing this just gets the value stored at memory location 0x5008. We know the array starts at 0x5000 and sizeof(int) is equal to 4. So (0x5008 - 0x5000) / 4 = 2. Which means that this entire expression is equivalent to the value of the second element of the array (zero-indexed), which is 0xd (13).

e) We know that *(p+5) is equivalent to p[5], and we know from the previous problem that something like (int*)(p[5]) is equivalent to an array starting at whatever value is stored in p[5], which is 0x4ffc. Since we know that the array starts at 0x5000, (0x4ffc - 0x5000) / 4 = -1, which means that this expression is equivalent to arr[-1], which is an error, since it is out of bounds of the array. In reality, the program could still continue to run or terminate immediately, but the behavior of the program from this expression onwards would be undefined.

5. Reverse! (Fall 2017 MT1 #3.1)

Fill in the blanks to complete the reverse function which takes in a head_ptr to the head of a linked list and returns a new copy of the linked list in reverse order. You must allocate space for the new linked list that you return. An example program using reverse is also shown below.

```
struct list_node {
    int val;
    struct list_node* next;
};

struct list_node* reverse( ______struct list node**______   head_ptr ) {
    struct list_node* next = NULL;
    struct list_node* ret;
    while (*head_ptr != NULL) {
        ret = _________malloc(sizeof(struct list_node))_____________ ;
        ret->val = _____________(*head_ptr)->val________________ ;
        ret->next = _____________next___________________ ;
        next = ______________ret____________________ ;
        *head_ptr = (*head_ptr)->next;
    }
    return _______________ret___________________;
}
```

[insert detailed solution here]