```c
//Dry part 1

#include <stdbool.h>
#include <stdlib.h>


typedef struct node_t {
    int x;
    struct node_t *next;
} *Node;
typedef enum {
    SUCCESS=0,
    MEMORY_ERROR,
    UNSORTED_LIST,
    NULL_ARGUMENT,
} ErrorCode;
int getListLength(Node list);

bool isListSorted(Node list);

/** The function receives two ordered lists and return a merged list.
\
 *
 * @param list1 - target list to be merged. If list1 is NULL nothing
will be done.
 * @param list2 - target list to be merged. If list2 is NULL nothing
will be done.
 * @param error_code - stores any errors the function made' or
success if the function merged successfully.

 * */
Node mergeSortedLists(Node list1, Node list2, ErrorCode* error_code);

/** The function receives a list and adds a new node, next to the
list head.
 *
 * @return-
 * MEMORY_ERROR if the allocation wasn't successful.
 * SUCCESS if otherwise.
 * */
ErrorCode nodeConcat(Node list, int num);


/** The function receives a linked list and frees it
 * @param list - the linked list to be freed. */
void freeLinkedList(Node list);

void addFinalNodes(Node ptr_list_src, Node ptr_list_dest, ErrorCode*
error_code);

Node mergeSortedLists(Node list1, Node list2, ErrorCode* error_code)
{
    if (list1 == NULL || list2 == NULL) {
        *error_code = NULL_ARGUMENT;
        return NULL;
    }
    if (!isListSorted(list2) || !isListSorted(list1)) {
        *error_code = UNSORTED_LIST;
        return NULL;
    }
    Node merged_list_head = malloc(sizeof(Node));
    if(merged_list_head == NULL) {
        *error_code = MEMORY_ERROR;
        return NULL;
    }
    Node ptr_list1 = list1, ptr_list2 = list2,  ptr_merged_list =
merged_list_head;

    while (ptr_list1!=NULL && ptr_list2!=NULL) {
        if(ptr_list1->x >= ptr_list2->x){
            *error_code = nodeConcat(ptr_merged_list, ptr_list1->x);
            if(*error_code == MEMORY_ERROR){
                freeLinkedList(merged_list_head);
                return NULL;
            }
            ptr_list1 = ptr_list1->next;
        }
        else {
            *error_code = nodeConcat(ptr_merged_list, ptr_list2->x);
            if (*error_code == MEMORY_ERROR) {
                freeLinkedList(merged_list_head);
                return NULL;
            }
            ptr_list2 = ptr_list2->next;
        }
        ptr_merged_list = ptr_merged_list->next;
    }

    addFinalNodes(ptr_list1, ptr_merged_list, error_code);
    if(*error_code == MEMORY_ERROR){
        freeLinkedList(merged_list_head);
        return NULL;
    }

    addFinalNodes(ptr_list2, ptr_merged_list, error_code);
    if(*error_code == MEMORY_ERROR){
        freeLinkedList(merged_list_head);
        return NULL;
    }
    *error_code = SUCCESS;
    return merged_list_head;
}


ErrorCode nodeConcat(Node list, int num){
    Node new_node = (Node) malloc(sizeof(Node));
    if(new_node == NULL)
        return MEMORY_ERROR;
    new_node->x = num;
    list->next = new_node;
    return SUCCESS;
}

void freeLinkedList(Node list){
    Node temp;
    while(list != NULL) {
        temp = list;
        list = list->next;
        free(temp);
    }
}
```

```
void addFinalNodes(Node ptr_list_src, Node ptr_list_dest, ErrorCode*
error_code) {
    while (ptr_list_src != NULL) {
        if (nodeConcat(ptr_list_dest, ptr_list_src->x) ==
MEMORY_ERROR) {
            *error_code = MEMORY_ERROR;
            freeLinkedList(ptr_list_dest);
            return;
        }
        ptr_list_dest->x = ptr_list_src->x;
        ptr_list_src = ptr_list_src->next;
    }
}
```

```
//Dry part 2
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *reversStringAndPrintByParity(char *str, int *length_ptr) //Both
foo and x are not meaningful names.
{ //Need to have its own line, by code conventions.
    if(length_ptr == NULL){
        return NULL;
    }
    char *reversed_str; //str2 is not a meaningful name.
    int i;
    *length_ptr = strlen(str); //Set the value of the integer pointer
instead of its address.
    reversed_str = malloc(*length_ptr + 1); //Extra 1 byte for the
null character ('\0').
    if (reversed_str == NULL){ //Memory error check was missing.
        return NULL;
    }
    for (i = 0; i < *length_ptr; i++) { //Missing Braces.
        reversed_str[i] = str[*length_ptr - i -1]; //Index mistake.
    }
    reversed_str[*length_ptr] = '\0'; //Missing null character ('\0')
at the end of reversed_str.
    if (*length_ptr % 2 != 0) { // Mistake (was '==' instead of
'!=').
        printf("%s", str);
    }
    if (*length_ptr % 2 == 0) { // Mistake (was '!=' instead of
'==').
        printf("%s", reversed_str);
    }
    return reversed_str;
}
```