Problem 1

CATCH THE BUG:
The following listings show some of the most common bugs when using pointers. Try to find them without executing the code. Note that some of them will not always cause a runtime error (sometimes they do, but sometimes they don't).

Deallocated pointers (1):

```
#include <iostream>
using namespace std;

int main() {
    int *p;
    if (true) {
        int x = 5;
        p = &x;
    }
    cout << *p << endl;
}
```

```
#include <iostream>
using namespace std;

int *getPtrToFive() {
    int x = 5;
    return &x;
}

int main() {
    int *p = getPtrToFive();
    cout << *p << endl;
}
```

Deallocated pointers (2):

```
#include <iostream>
using namespace std;

int main() {
    int *x = new int(4);
    delete x;
    cout << *x << endl;
}
```

```
#include <iostream>
using namespace std;

int main() {
    int *x = new int(4);
    int *y = x;
    delete x;
    delete y;
}
```

```
#include <iostream>
using namespace std;

int main() {
    int *x = new int(4);
    int *y = x;
    delete x;
    cout << *y << endl;
}
```

Problem 1 [Deallocated pointers (2):] continued on next page. . .

Leaked memory:

```
#include <iostream>
using namespace std;

int main() {
    int *p;
    for (int i = 0; i < 3; ++i) {
        p = new int(i);
        cout << *p << endl;
    }
}
```

```
#include <iostream>
using namespace std;

int main() {
    int *p;
    for (int i = 0; i < 3; ++i) {
        p = new int(i);
        cout << *p << endl;
    }
    delete p;
}
```

```
#include <iostream>
using namespace std;

struct Node {
    int val;
    Node *next;
};

int main() {
    Node *head = new Node;
    head->next = new Node;
    delete head;
}
```

```
#include <iostream>
using namespace std;

int main() {
    int *p = new int(3);
    p = NULL;
    delete p;
}
```

Uninitialized and non-dynamic memory:

```
#include <iostream>
using namespace std;

int main() {
    Node *head;
    cout << head->value << endl;
}
```

```
#include <iostream>
using namespace std;

int main() {
    int x = 5;
    int *p = &x;
    cout << *p << endl;
    delete p;
}
```

Problem 1 continued on next page. . .

Reference and pointer parameters:

```
#include <iostream>
using namespace std;

void swap(int a, int b) {
    int aux = a;
    a = b;
    b = aux;
}

int main() {
    int x = 5;
    int y = 7;
    swap(x, y);
    cout << x << " " << y << endl;
}
```

Problem 2

SINGLY-LINKED LISTS:
In this problem we will consider the classic singly linked list structure:

- a single head pointer points to the first node in the list (the empty list is represented by a NULL head pointer);
- each node is a tuple with two fields: the *data* value, and a single *next* pointer to the next node;

```
struct Node {
    int data;
    Node *next;
};
```

- the *next* pointer of the last node is NULL.

The following code creates a list with four nodes having values 1, 2, 3, and 4, respectively.

Problem 2 continued on next page. . .

Listing 1: mylist.h

```
#ifndef MYLIST_H
#define MYLIST_H

struct Node {
    int data;
    Node *next;
};

void push(Node **headRef, int newValue);
#endif
```

Listing 2: mylist.cpp

```
#include "mylist.h"

void push(Node **headRef, int newValue) {
    Node *n = new Node;
    n->next = *headRef;
    n->data = newValue;
    *headRef = n;
}
```

Listing 3: testlist.cpp

```
#include <iostream>
#include "mylist.h"
using namespace std;

int main() {
    Node *head = NULL; // empty list
    for (int i = 4; i > 0; --i) push(&head, i);
}
```

Modify mylist.h and mylist.cpp to include each of the following functionalities. Test them modifying the testlist.cpp file.

count

Write a count function that counts the number of times a given int occurs in a list.

```
int count(Node *head, int searchFor) {
     // Your code here
}
```

getNth

Write a getNth function that takes a linked list and an integer index and returns the data value stored in the node at that index position. We will use the usual C++ numbering convention that the first node is index 0, the second is index 1, and so on. In case the index is not valid, the function should return -1.

```
int getNth(Node *head, int index) {
    // Your code here
}
```

deleteList

Write a function deleteList that takes a list, deallocates all of its memory and sets its head pointer to the empty list.

```
void deleteList(Node **headRef) {
    // Your code here
}
```

pop

Write a pop function that takes a non-empty list, deletes the head node, and returns the head node's data.

```
int pop(Node **headRef) {
    // Your code here
}
```

insertNth

Write a function insertNth that inserts a new node at a given index within a list. The index is in the range $[0, \ldots, length]$, and the new node should be inserted so as to be at that index.

```
void insertNth(Node **headRef, int index, int newValue) {
    // Your code here
}
```

Problem 2 continued on next page. . .

append

Write a function append that takes two lists lst1 and lst2, appends lst2 onto the end of lst1, and then sets lst2 to the empty list.

```
void append(Node **lst1Ref, Node **lst2Ref) {
    // Your code here
}
```

sortedInsert

Write a function sortedInsert that given a list that is sorted in increasing order, and a single node, inserts the node into the correct sorted position in the list.

```
void sortedInsert(struct node** headRef, struct node* newNode) {
    // Your code here
}
```

reverse

Write an iterative function reverse that reverses a list by rearrenging all the *next* pointers and the head pointer. Ideally, reverse should only need to make one pass of the list.

```
void reverse(Node **headRef) {
    // Your code here
}
```

Bonus: Try a recursive solution to this problem.