CS 010C - Assignment 1
Andrew Pham
Collaborated with Arjun Bhalla, Rushil Shah, and Devesh Panda

Problem 1:

(a)

```
void List::reverse() {
    Node *prevNode = nullptr;
    Node *currNode = head;
    Node *nextNode = nullptr;
    while(currNode != nullptr) {
        nextNode = currNode->next;
        currNode->next = prevNode;
        prevNode = currNode;
        currNode = nextNode;
    }
    Node *temp = head;
    head = tail;
    tail = temp;
}
```

(b)
We start by creating 3 Node pointers prevNode, currNode, and nextNode, setting currNode equal to head and prevNode and nextNode equal to nullptr.

Within the while loop, nextNode is set to currNode->next, in order to hold the address of the Node following currNode. Because we are changing what currNode->next points to, this step is important so that we can continue to traverse the list.

We proceed to make currNode's next pointer point to prevNode, which reverses currNode's next pointer. We then move the pointers one position ahead, setting prevNode to currNode, and setting currNode to nextNode. The while loop repeats this process until currNode reaches the end of the list. The next pointers of each Node in the list now point to the Node that originally preceded it.

The Node that head points to has become the new tail Node, and the Node that tail points to has become the new head Node. After performing a simple swap to switch the head and tail pointers, we have fully reversed the linked list.

Problem 2:

(a)
```
void List::remove_duplicates() {
    Node *currNode = head;
    while(currNode != nullptr && currNode->next != nullptr) {
        if(currNode->data == currNode->next->data) {
            Node *nextNode = currNode->next;
            currNode->next = currNode->next->next;
            delete nextNode;
        }
        else {
            currNode = currNode->next;
        }
    }
    tail = curr;
}
```

(b)
We begin by creating a Node pointer currNode, which we set as equal to head. In the condition for our while loop, we have currNode != nullptr, which is the usual condition for traversing a linked list, but we also need currNode->next != nullptr. This is because we need to check the value at the Node after currNode, to see if it is a duplicate.

Our if statement executes if currNode's data is equal to currNode->next's data - if the Node after currNode is a duplicate. First, a Node pointer nextNode is set equal to the Node after currNode. Then, currNode's next pointer is set to point to currNode->next->next, the Node after the next node. This basically removes the duplicate Node, as it is no longer pointed to by currNode's next pointer. To avoid a memory leak, we then use the delete operator on nextNode, freeing the memory stored by the duplicate Node.

If currNode->next is NOT a duplicate, then currNode is simply set to the next Node. We continue to traverse the list, removing duplicates when they appear. After exiting the while loop, in order to avoid a potential dangling pointer in case the last Node was a duplicate and had been deleted, we set tail equal to curr, which should be pointing to the last Node in the list in all cases.

Problem 3:

Output should be:
```
100 116 148
200 202 206
300 332 396
```

The problem states that our fancy computer stores type char with 16 bits, type int with 128 bits, and type double with 256 bits. We must remember that a byte contains 8 bits. So in essence, the problem is saying that our fancy computer stores type int with 16 bytes, type char with 2 bytes, and type double with 32 bytes.

A C++ array stores its elements contiguously, so the numerical distance between the addresses of each successive element in memory should be equal to the number of bytes stored by the element's data type. So, the difference between each successive element in an array should be 16, 2, and 32 for int, char, and double, respectively for our fancy computer.

```
HtoI(A+1) = 100 + 16 = 116       HtoI(A+3) = 100 + 16(3) = 148
HtoI(B+1) = 200 + 2 = 202        HtoI(B+3) = 200 + 2(3) = 206
HtoI(C+1) = 300 + 32 = 332       HtoI(C+3) = 300 + 32(3) = 396
```

Problem 4:

(a)
The code should output the following:

```
20
10
```

(b)
In ClassA's constructor, the "new" operator is used twice to dynamically allocate an int on the heap, storing the addresses of the newly allocated memory in the int pointer variables p1 and p2.

In main(), creating an object of ClassB will invoke ClassA's constructor to create an object of ClassA, because ClassB has a private member of type ClassA, called a_obj. So, instantiating an object of type ClassB dynamically allocates memory to the heap.

In main(), an object of ClassB, b_obj, is instantiated, which invokes ClassA's constructor and dynamically allocates two ints on the heap. When main() ends, the pointer variables of b_obj.a_obj, which hold the addresses of the dynamically allocated ints, go out of scope.

The operating system cannot locate the addresses of the dynamically allocated ints after those pointers go out of scope, and there is no destructor in ClassA or ClassB that frees the dynamically allocated memory using the delete keyword. Therefore, a memory leak occurs.