CS 32 Worksheet 1

This worksheet is entirely **optional**, and meant for extra practice. Some problems will be more challenging than others and are designed to have you apply your knowledge beyond the examples presented in lecture, discussion or projects. All exams will be done on paper, so it is in your best interest to practice these problems by hand and not rely on a compiler.

Concepts

Copy constructors, assignment operators, singly-linked lists

Problems

1) What is the output of the following code?

```
class A {
public:
      A()
      { cout << "DC" << endl; }
      A(const A& other)
      { cout << "CC" << endl; }
      A& operator=(const A& other)
      { cout << "AO" << endl; return *this; }
      ~A()
      { cout << "Destructor!" << endl;}
};

int main() {
      A arr[3];
      arr[0] = arr[1];
      A x = arr[0];
      x = arr[1];
      A y(arr[2]);
      cout << "DONE" << endl;
}
```

2) Complete the copy constructor, assignment operator, and destructor of the following class. Be careful to avoid aliasing, memory leaks, and other pointer issues!

```
class A {
public:
```

```
  A(int sz) { //...constructor code }

  A(const A& other) {
    //...implement this!
  }

  A& operator=(const A& other) {
    //...implement this!
  }

  //...other functions

  ~A() {
    //...implement this!
  }

private:
  //one dynamically allocated B object; assume B has a default
  //constructor, a copy constructor, and an assignment operator
  B* b;
  //dynamically allocated array
  int* arr;
  //size of arr (determined by a constructor)
  int n;
  string str;
};
```

3) Find the **4 errors** in the following class definitions so the main function runs correctly.

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

class Account {
public:
  Account(int x) {
    cash = x;
  }
  int cash;
}

class Billionaire {
public:
```

```
  Billionaire(string n){
    account = Account(10000);
    offshore = Account(1000000000);
    name = n;
  }

  Account account;
  Account* offshore;
  string name;
};

int main() {
  Billionaire jim = Billionaire("Jimmy");
  cout << jim.name << " has " <<
       << jim.account.cash + jim.offshore->cash << endl;
}

Output: Jimmy has 1000010000
```

4) After being defined by the above code, Jim the Billionaire funded a cloning project and volunteers himself as the first human test subject. Sadly, all his money isn't cloned, so his clone has his name, but has $0. Add the needed function so the following main function produces the following output.

```
int main() {
  Billionaire jim = Billionaire("Jimmy");
  Billionaire jimClone = jim;
  cout << jimClone.name << " has " <<
       << jimClone.account.cash + jimClone.offshore->cash
       << endl;
  cout << jim.name << " has " <<
       << jim.account.cash + jim.offshore->cash << endl;
}

Output:    Jimmy has 0
           Jimmy has 1000010000
```

5) Write a function `cmpr` that takes in a linked list and an array and returns the largest index up to which the two are identical. The function should return -1 if the first element of the list and array are not identical.

```
Function declaration: cmpr(Node* head, int *arr, int arr_size);

// head -> 1 -> 2 -> 3 -> 5 -> 6
```

```
int a[6] = {1, 2, 3, 4, 5, 6};
cout << cmpr(head, a, 6); // Should print 2

int b[7] = {1, 2, 3, 5, 6, 7, 5};
cout << cmpr(head, b, 7); // Should print 4

int c[3] = {5, 1, 2};
cout << cmpr(head, c, 3); // Should print -1

int d[3] = {1, 2, 3};
cout << cmpr(head, d, 3); // Should print 2
```

6) Class LL contains a single member variable - a pointer to the head of a singly linked list. Using the definitions for class LL and a Node of the linked list, implement a copy constructor for LL. The copy constructor should create a new linked list with the same number of nodes and same values.

```
class LL {
public:
      LL() { head = nullptr; }
private:
      struct Node {
      public:
            int val;
            Node* next;
      }

      Node* head;
}
```

7) Using the same class LL from the previous problem, write a function *findNthFromLast* that returns the value of the Node that is nth from the last Node in the linked list.

```
int findNthFromLast(int n, LL list);
```

`findNthFromLast(2, head)` should return 3 when given the following linked list:

head -> 1 -> 2 -> 3 -> 4 ->5->nullptr

If the nth from the last Node does not exist, *findNthFromLast* should return -1.

You may assume all values that are actually stored in the list are nonnegative.

8) Suppose you have a struct **Node** and a class **LinkedList** defined as follows:

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

class LinkedList {
public:
      void rotateLeft(int n); //rotates head left by n
      //Other working functions such as insert and printItems
private:
      Node* head;
}
```

Give a definition for the *rotateLeft* function such that it rotates the linked list represented by *head* left by *n*. Rotating a list left consists of shifting elements left, such that elements at the front of the list loop around to the back of the list. The new start of the list should be stored within *head*.

Ex: Suppose you have a **LinkedList** object *numList*, and printing out the values of *numList* gives the following output, with the head pointing to the node with 10 as its value:
10 -> 1 -> 5 -> 2 -> 1 -> 73
Calling *numList*.rotateLeft(3) would alter *numList*, so that printing out its values gives the following, new output, with the head storing 2 as its value:
2 -> 1 -> 73 -> 10 -> 1 -> 5

The *rotateLeft* function should accept only integers greater than or equal to 0. If the input does not fit this requirement, it may handle the case in whatever reasonable way you desire.

9) Suppose you have a struct **Node** and a class **LinkedList** defined as they were in problem 8.

Give a definition for the *rotateRight* function such that it rotates the linked list represented by *head* right by *n*. Rotating a list right is similar to rotating it left, but it consists of shifting elements right, such that elements at the end of the list loop back to the front of the list. The new start of the list should be pointed to by *head*.

Ex: Suppose you have a **LinkedList** object *numList*, and printing out the values of *numList* gives the following output, with the head storing 3 as its value:
3 -> 4 -> 7 -> 10 -> 1 -> 4
Calling *numList*.rotateRight(4) would alter *numList*, so that printing out its values gives the following, new output, with the head storing 7 as its value:
7 -> 10 -> 1 -> 4 -> 3 -> 4

The *rotateRight* function should accept only integers greater than or equal to 0. If the input does not fit this requirement, it may handle the case in whatever reasonable way you desire.

10) Given a sorted linked list, write a function that guarantees insertion of a value in a **sorted way**.

    The function header is given as:
    `void sortedInsert(Node*& head_ref, Node* new_node)`

    For example, if the linked list is 2 -> 3 -> 6 -> 10 and the given value is 8, then after calling your function, the list should be 2 -> 3 -> 6 -> 8 -> 10

    This is the implementation of each node:
```
/* Link list node */
struct Node
{
    int data;
    Node* next;
};
```

11) The function in problem 10 would work correctly in most cases even if its first parameter were declared as a `Node*` instead of a `Node*&`. Under what circumstances would it work incorrectly if that parameter were declared as a `Node*`, yet correctly if it were declared as `Node*&`?

12) Given two linked lists where every node represents a character in the word. Write a function compare() that works similar to strcmp(), i.e., it returns 0 if both strings are same, a positive integer if the first linked list is lexicographically greater, and a negative integer if the second string is lexicographically greater.

    The header of your function is given as:
    `int compare(Node* list1, Node* list2)`

Example:

```
If list1 = a -> n -> t
   list2 = a -> r -> k
then compare(list1, list2) < 0
```

13) What is the output of the following code:

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

class B {
     int m_val;
public:
     B(int x): m_val(x) { cout << "Wow such " << x << endl; }
     B(const B& other) {
          cout << "There's another me???" << endl;
          m_val = other.m_val;
     }
     ~B() {
          cout << "Twas a good life" << endl;
     }
};

class A {
     int m_count;
     B* m_b;
public:
     A(): m_count(9.5) {
          cout << "Construct me with " << m_count << endl;
          m_b = new B(m_count+10);
     }
     A(const A& other) {
          cout << "Copy me" << endl;
          m_count = other.m_count;
          m_b = other.m_b != NULL ? new B(*other.m_b) : NULL;
     }
     ~A() {
          cout << "Goodbye cruel world" << endl;
          if (m_b)
               delete m_b;
     }
     int getCount() { return m_count; }
};
```

```
int main() {
     A a1, a2;
     A a3 = a2;
     B b1(a3.getCount());
     cout << "Where are we?" << endl;
}
```

14) Write a function that takes in the head of a singly linked list, and returns the head of the linked list such that the linked list is reversed.
Example:
Original: LL = 1 → 2 → 3 → 4 → 5
Reversed: LL = 5 → 4 → 3 → 2 → 1

We can assume the Node of the linked list is implemented as follows:

```
/* Link list node */
struct Node
{
    int data;
    Node* next;
};

Node* reverse(Node* head) {
     // Fill in this function
}
```

15) Write a function `combine` that takes in two sorted linked lists and returns a pointer to the start of the resulting combined sorted linked list.

```
// head -> 1 -> 3 -> 6 -> 9
// head2 -> 7 -> 8 -> 10
// Node* combine(Node* h, Node* h2);
Node* res = combine(head, head2);
// res -> 1 -> 3 -> 6 -> 7 -> 8 -> 9 -> 10
```