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.

Solutions are written in red. The solutions for **programming** problems are not absolute, it is okay if your code looks different; this is just one way to solve the specific problem.

If you have any questions or concerns please email raykwan@ucla.edu, or go to any of the LA office hours.

Concepts

Copy constructors, assignment operators, 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;
}
```

Difficulty: Easy

Output:
DC
DC
DC
AO
CC
AO
CC
DONE
Destructor!
Destructor!
Destructor!
Destructor!
Destructor!

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) {
    //...implement this!
  }

  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;
};
```

Difficulty: Medium

```
class A {
public:
  A(int sz) {
    b = new B;
    arr = new int[sz];
    n = sz;
  }

  A(const A& other) {
    b = new B(*other.b);
    n = other.n;
    arr = new int[n];
    for (int i = 0; i < n; i++) {
      arr[i] = other.arr[i];
    }
    str = other.str;
  }

  A& operator=(const A& other) {
    if (this != &other) {
      delete b;
      delete [] arr;
      b = new B(*other.b);
      n = other.n;
      arr = new int[n];
      for (int i = 0; i < n; i++) {
        arr[i] = other.arr[i];
      }
      str = other.str;
    }
    return *this;
  }
```

```
    //...other functions

    ~A() {
      delete b;
      delete [] arr;
    }

  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(10000){
    account = Account(10000);
    offshore = new Account(1000000000);
    name = n;
  }

  ~Billionaire() {
     delete offshore;
  }
```

```
  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
```

Difficulty: Medium

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 to the Billionaire class 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;
}

  Billionaire(const Billionaire &b)
   : account(0), name(b.name)
  {
    offshore = new Account(0);
  }
```

```
Output:    Jimmy has 0
           Jimmy has 1000010000
```

Difficulty: Easy

5) 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 != nullptr) ? new B(*other.m_b) :
	nullptr;
	}
	~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;
}
```

Difficulty: Easy

Output:

Construct me with 9
Wow such 19
Construct me with 9
Wow such 19
Copy me
There's another me???
Wow such 9
Where are we?
Twas a good life
Goodbye cruel world
Twas a good life
Goodbye cruel world
Twas a good life
Goodbye cruel world
Twas a good life

6) 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
```

Difficulty: Easy

```
int cmpr(Node *head, int *arr, int arr_size) {
	int i = 0;
	Node *curr = head;
	int index = -1;
	while (i < arr_size && curr != NULL && curr->data ==
	arr[i]) {
		index++;
		i++;
		curr = curr->next;
	}
	return index;
}
```

7) 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; }

	LL(const LL& other) {
	  if (other.head == nullptr)
	    head = nullptr;
	  else {
	    head = new Node;
	    head->val = other.head->val;
	    head->next = other.head->next;

	    Node* thisCurrent = head;
	    Node* otherCurrent = other.head;
	    while (otherCurrent->next != nullptr) {
	      thisCurrent->next = new Node;
	      thisCurrent->next->val = otherCurrent->next->val;
	      thisCurrent->next->next = otherCurrent->next->next;

	      thisCurrent = thisCurrent->next;
	      otherCurrent = otherCurrent->next;
	    }
	  }
	}
```

```
private:
      struct Node {
      public:
            int val;
            Node* next;
      };

      Node* head;
};
```

Difficulty: Easy

8) 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 LL::findNthFromLast(int n);
```

`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.

Difficulty: Medium

```
int LL::findNthFromLast(int n) {
  Node* p = head;
  for (int i = 0; i < n; i++) {
    if (p == nullptr) {
      return -1;
    }
    p = p->next;
  }
  if (p == nullptr) {
    return -1;
  }

  Node* nthFromP = head;
  while (p->next != nullptr) {
```

```
    p = p->next;
    nthFromP = nthFromP->next;
  }
  return nthFromP->val;
}
```

9) 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.

Difficulty: Medium

```
void LinkedList::rotateLeft(int n) {
  if (head == nullptr)
    return;
```

```
    int size = 1;
    Node* oldTail = head;
    while (oldTail->next != nullptr) {
      size++;
      oldTail = oldTail->next;
    }

    if (n % size > 0) {
      int headPos = n % size;
      Node* newTail = head;
      for (int x = 0; x < headPos - 1; x++) {
        newTail = newTail->next;
      }
      Node* newHead = newTail->next;

      newTail->next = nullptr;
      oldTail->next = head;
      head = newHead;
    }
  }
```

10) 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 the 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
```

Difficulty: Medium

```
int compare(Node *list1, Node *list2)
     {
         // Traverse both lists. Stop when either end of a linked
         // list is reached or current characters don't match
         while (list1 && list2 && list1->c == list2->c)
```

```
    {
        list1 = list1->next;
        list2 = list2->next;
    }

    //  If both lists are not empty, compare mismatching
    //  characters
    if (list1 && list2)
        return (list1->c > list2->c)? 1: -1;

    // If either of the two lists has reached end
    if (list1 && !list2) return 1;
    if (list2 && !list1) return -1;

    // If none of the above conditions is true, both
    // lists have reached end
    return 0;
}
```

11) 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
}
```

Difficulty: Medium

```
// The idea here is to reverse each node one step at
// a time with a previous and current pointer
// At the end prev should point to the last element in the
// original linked list
```

```
Node* reverse(Node* head) {
      Node* prev = nullptr;
      Node* curr = head;
      while (curr != nullptr) {
            Node* next = head->next;
            head->next = prev;
            prev = head;
            head = next;
      }
      return prev;
}
```

12) 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. You may write a helper function to call in your function `combine`.

```
// 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
```

Difficulty: Medium

```
Node* combine(Node* h, Node* h2) {
      if (h == nullptr) {
            return h2;
      }
      if (h2 == nullptr) {
            return h;
      }

      ListNode* newList;

      if (h->val <= h2->val) {
            newList = h;
            h = h->next;
      }
      else {
            newList = h2;
            h2 = h2->next;
      }
```

```
    ListNode* newNext = newList;

    while (h != nullptr && h2 != nullptr) {
        if (h->val <= h2->val) {
                newNext->next = h;
                h = h->next;
        }
        else {
                newNext->next = h2;
                h2 = h2->next;
        }
        newNext = newNext->next;
    }

    if (h != nullptr) {
        newNext->next = h;
    }
    else if (h2 != nullptr) {
        newNext->next = h2;
    }

    return newList;
}
```