CS 32 Solutions Week 5

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. Although exams are online this quarter, it is still in your best interest to practice these problems by hand and not rely on a compiler.

If you have any questions or concerns please contact your LA or go to any of the LA office hours.

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.

Concepts: Inheritance, Polymorphism, Recursion

Inheritance, Polymorphism

1. What does the following code output, and what changes do you have to make to it to have it output “`I’m Gene`“?

   `Current output: I’m a living thing`

   HINT: You will need to use the virtual keyword!

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

class LivingThing {
  public:
     virtual void intro() { cout << "I’m a living thing" <<
endl; }
};

class Person : public LivingThing {
  public:
       // repeating the word virtual is not required here (but
       // recommended as a reminder to a human reader)
     void intro() { cout << "I’m a person" << endl; }
};
```

```
class UniversityAdministrator : public Person {
  public:
        // repeating the word virtual is not required here (but
        // recommended as a reminder to a human reader)
     void intro() {
          cout << "I’m a university administrator" << endl;
     }
};

class Chancellor : public UniversityAdministrator {
  public:
        // repeating the word virtual is not required here (but
        // recommended as a reminder to a human reader)
     void intro() { cout << "I’m Gene" << endl; }
};

int main() {
     LivingThing* thing = new Chancellor();
     thing->intro();
     ...
}
```

2. What is the output of the following program?

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

class Pet {
  public:
     Pet() { cout << "Pet" << endl; }
     ~Pet() { cout << "~Pet" << endl; }
};

  // This is an unusual class that derives from Pet but also
  // contains a Pet as a data member.
class Dog : public Pet {
  public:
     Dog() { cout << "Woof" << endl; }
     ~Dog() { cout << "Dog ran away!" << endl; }
  private:
     Pet buddy;
};
```

```
int main() {
      Pet* milo = new Dog;
      delete milo;
}
```

```
Pet
Pet
Woof
~Pet
```

Undefined behavior after this, because Pet's destructor is not declared virtual.

3. Suppose the class declaration for Pet was changed as shown below. What is the new output of the code in problem 2) with these new changes?

```
class Pet {
  public:
      Pet() { cout << "Pet" << endl; }
      virtual ~Pet() { cout << "~Pet" << endl; }
};
```

```
Pet
Pet
Woof
Dog ran away!
~Pet
~Pet
```

4. Would the following work in C++? Why or why not?

```
class B;

class A : public B { … code for A … };
class B : public A { … code for B … };
```

Conceptually, this code is saying "A is a proper subset of B, and B is a proper subset of A", which is nonsense.

Practically, every object of a derived class contains an instance of the base class. If the code above were legal, a B object would contain an A object that contains a B object that contains an A object, ad infinitum.

Recursion

1. Given a singly-linked list class LL with a member variable *head* that points to the first *Node* struct in the list, write a function to recursively delete the whole list, void LL::deleteList().. Assume each Node object has a next pointer.

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

class LL {
  public: // other functions such as insert not shown
     void deleteList(); // implement this function
  private: // additional helper allowed
     Node* m_head;
};
```

```
void LL::deleteListHelper(Node* &head) {
  if (head == nullptr)
      return;

  deleteListHelper(head->next);
  delete head;
  head = nullptr;
}

void LL::deleteList() {
  deleteListHelper(m_head);
}
```

2. Implement the function `isPalindrome` recursively. The function should return whether the given string is a palindrome. A palindrome is described as a word, phrase or sequence of characters that reads the same forward and backwards.

```
bool isPalindrome(string foo);
```

```
    isPalindrome("kayak"); // true
    isPalindrome("stanley yelnats"); // true
    isPalindrome("LAs rock"); // false (but the sentiment is true
:))

    bool isPalindrome(string foo) {
        int len = foo.length();
        if (len <= 1)
            return true;
        if (foo[0] != foo[len-1])
            return false;
        return isPalindrome(foo.substr(1, len-2));
    }
```

3. Write a recursive function `isPrime` to determine whether a given positive integer input is a prime number or not. You may add an auxiliary helper function if necessary.

    Example:

        isPrime(11) → true
        isPrime(4) → false

```
// We notice that without a secondary parameter to keep
count of
// where we are in our check, this problem is impossible
// recursively. Thus the solution can be done with either
a
// default parameter or with an auxiliary helper function.

bool isPrime(int num) {
    return isPrimeHelper(num, 2); // start with testing
                              // divisibility by 2
}
bool isPrimeHelper(int num, int i) {
    if (num <= 2)
        return num == 2; // 1 is not prime
    if (num % i == 0)  // not prime if divisible by i
        return false;
    if (i*i > num)  // is prime if exhausted all divisors
        return true;
    return isPrimeHelper(num, i + 1); // increment i and
see if it is divisible by it
}
```

4. Implement the following recursive function:

   ```
   string longestCommonSubsequence(string s1, string s2);
   ```

   The function should return the longest common subsequence of characters between the two strings s1 and s2. Basically, it should return a maximum length string of characters that are common to both strings and are in the same order in both strings.

   Example:

   ```
   string res = longestCommonSubsequence(“smallberg”,
   “nachenberg”);
   //res should contain “aberg” as seen in the green chars
   res = longestCommonSubsequence(“los angeles”,
   “computers”);
   //res should contain the string “oes”
   ```

   ```
   string longestCommonSubsequence(string s1, string s2) {
        if (s1.empty()  ||  s2.empty()) // base case: either
   empty
        return "";
        // split the strings into head and tail for
   simplicity
        char s1_head = s1[0];
        string s1_tail = s1.substr(1);
        char s2_head = s2[0];
        string s2_tail = s2.substr(1);

        // if heads are equal, use the head and
        // recursively find rest of common subsequence
        if (s1_head == s2_head)
        return s1_head + longestCommonSubsequence(s1_tail,
   s2_tail);
        // heads different, so check for common subsequences
   not
        // including one of the heads
        string if_behead_s1 =
   longestCommonSubsequence(s1_tail, s2);
   ```

```
        string if_behead_s2 = longestCommonSubsequence(s1,
    s2_tail);

        // return the longer of the subsequences we found
        return if_behead_s1.length() >= if_behead_s2.length()
    ? if_behead_s1 : if_behead_s2;
    }
```

Additional Practice Problems

Inheritance, Polymorphism

1. Given the following class declarations, complete the implementation of each constructor so that the program compiles. Your implementations should correctly assign constructor arguments to class member variables.

   HINT: You will need to use initializer lists!

```
class Animal {
public:
        Animal(string name);
private:
        string m_name;
};

class Cat : public Animal {
public:
        Cat(string name, int amountOfYarn);
private:
    int m_amountOfYarn;
};

class Himalayan : public Cat {
public:
    Himalayan(string name, int amountOfYarn);
};

class Siamese: public Cat {
    public:
    Siamese(string name, int amountOfYarn, string toyName);
private:
      string m_toyName;
};

Animal::Animal(string name)
      : m_name(name) {}

Cat::Cat(string name, int amountOfYarn)
      : Animal(name), m_amountOfYarn(amountOfYarn) {}

Himalayan::Himalayan(string name, int amountOfYarn)
: Cat(name, amountOfYarn) {}

Siamese::Siamese(string name, int amountOfYarn, string toyName)
: Cat(name, amountOfYarn), m_toyName(toyName) {}
```

2. The following code has several errors. Rewrite the code so that it can successfully compile. Try to catch the errors without using a compiler!

```
class LivingThing {
public:
      LivingThing(int a) { age = a; }
      void myBirthday() { age++; }
private:
      int age;
};

class Person : public LivingThing {
public:
      Person(int a) : LivingThing(a) { ~~age = a;~~ }
      void birthday() {
            ~~age++;~~
            myBirthday();
      }
};
```

3. Examine the following code and determine its output.

```cpp
#include <iostream>
#include <string>

using namespace std;

class A {
public:
     A() : m_val(0) {
          cout << "What a wonderful world! " << m_val << endl;
     }
     virtual ~A() { cout << "Guess this is goodbye " << endl; }
     virtual void saySomething() = 0;
     virtual int giveMeSomething() = 0;
private:
     int m_val;
};

class B : public A {
public:
     B() : m_str("me"), m_val(1) {
          cout << m_str << " has just been birthed."  << endl;
     }
     B(string str, int val) : m_str(str), m_val(val) {
          cout << "More complex birth " << m_str << endl;
     }
     ~B() {
          cout << "Why do I have to leave this world!" << endl;
     }
     virtual void saySomething() {
          cout << "Coming in from " << m_str << " with "
               << giveMeSomething() << endl;
     }
     virtual int giveMeSomething() { return m_val*5; }
private:
     int m_val;
     string m_str;
};

class C {
public:
     C() : m_val(2) {
          m_b = new B("C", m_val);
          cout << "Hello World!!" << endl;
     }
     C(const B& b, int val) :  m_val(val) {
            m_b = new B(b);
            cout << m_b->giveMeSomething() << endl;
      }
      ~C() {
            m_b->saySomething();
            delete m_b;
            cout << "Goodbye world!" << endl;
      }
private:
      B* m_b;
      int m_val;
};

int main() {
      B* b_arr = new B[3];
      for(int i = 0; i < 3; i++) {
            b_arr[i].saySomething();
      }
      B b("B", 5);
      A* a = &b;
      cout << a->giveMeSomething() << endl;
      C c;
      C c2(b, b.giveMeSomething());
      delete [] b_arr;
}
```

```
What a wonderful world! 0
me has just been birthed.
What a wonderful world! 0
me has just been birthed.
What a wonderful world! 0
me has just been birthed.
Coming in from me with 5
Coming in from me with 5
Coming in from me with 5
What a wonderful world! 0
More complex birth B
25
What a wonderful world! 0
More complex birth C
Hello World!!
25
Why do I have to leave this world!
Guess this is goodbye
```

```
Why do I have to leave this world!
Guess this is goodbye
Why do I have to leave this world!
Guess this is goodbye
Coming in from B with 25
Why do I have to leave this world!
Guess this is goodbye
Goodbye world!
Coming in from C with 10
Why do I have to leave this world!
Guess this is goodbye
Goodbye world!
Why do I have to leave this world!
Guess this is goodbye
```

Recursion

1. What does the following code output and what does the function LA_power do?

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

     int LA_power(int a, int b)
     {
        if (b == 0)
          return 0;
        if (b % 2 == 0)
          return LA_power(a+a, b/2);

        return LA_power(a+a, b/2) + a;
     }

     int main()
     {
       cout << LA_power(3, 4) << endl;
     }
```

It outputs 12. LA_power returns the result of multiplying its arguments.

2. Implement the recursive function `merge` that merges two sorted linked lists `l1` and `l2` into a single sorted linked list. The lists are singly linked; the last node in a list has a null next pointer. The function should return the head of the merged linked list. No new Nodes should be allocated while merging.

   Example:

   ```
   l1:   1 -> 4 -> 6 -> 8
        l2:   3 -> 9 -> 10
   ```

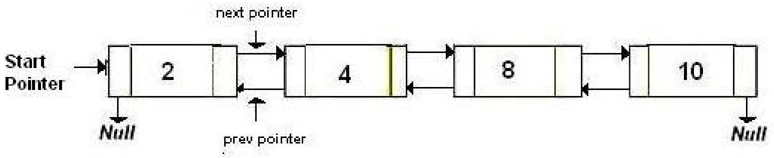
   After merge: `1 -> 3 -> 4 -> 6 -> 8 -> 9 -> 10`

```
     // Node definition for singly linked list
     struct Node {
          int val;
          Node* next;
     };

     Node* merge(Node* l1, Node* l2) {
          // base cases: if a list is empty, return the other list
          if (l1 == nullptr)
          return l2;
          if (l2 == nullptr)
          return l1;

     // determine which head should be the head of the merged list
     // then set head->next to the head returned from recursive
calls
          Node* head;
          if (l1->val < l2->val) {
          head = l1;
          head->next = merge(l1->next, l2);
          }
          else {
          head = l2;
          head->next = merge(l1, l2->next);
          }

          // return the head of the merged list
          return head;
     }
```

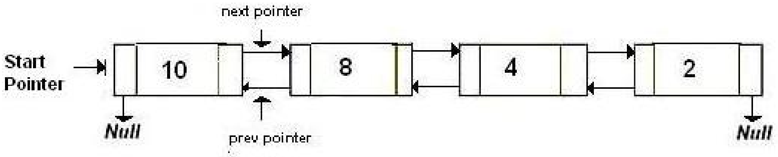
3. Implement `reverse`, a recursive function to reverse a doubly linked list. It returns a pointer to the new head of the list. The integer value in each node must not be changed (but of course the pointers can be).

Example:

Original:



After:



```
// Node definition for doubly linked list
struct Node {
      int val;
      Node* next;
      Node* prev;
};

Node* reverse(Node* head) {
      if (head == nullptr)
            return head;
      // Swap next and prev
      Node* temp = head->next;
      head->next = head->prev;
      head->prev = temp;
      // If previous is null then we are done
      if (head->prev == nullptr)
            return head;
      return reverse(head->prev);
}
```