CSE2050 - Programming in a Second Language
Assignment 4: Template Classes

October 21, 2017

1 Submission

- Due Date: **November 7th, 11:59pm**
- Submit on **Canvas**.

2 Description

Once again, we revisit the problem discussed in the previous assignments. The goal this time is to generalize the concept of computing the "largest" common value to cover strings, as well as numbers. We will use the GCD as the common value when dealing with lists of numbers , and the Longest Common Substring (or LCS) as the common value for strings (not to be confused with the *Longest common subsequence*).

Using a *template class*, we can implement a "generic" class for the linked list (and the nodes) that provides all general functionalities (like adding node, deleting node,...etc) without being dependent on the data type used.

3 Implementation details

3.1 Notes on the Longest Common Substring (LCS)

- LCS is the longest consecutive (uninterrupted) string that is a sub-string of two or more strings.
- For example, the LCS of these two strings "AA**BAABC**BBABCA" and "C**BAABC**AABCC" is "**BAABC**", and has a length of 5.
- While performance is not a grading criterion for this assignment, there are very inefficient methods (like *brute force*) to solve this problem that might require more execution time or memory space for larger test cases than our server allows. We recommend a dynamic programming approach to solve this problem (linked below).

3.2 Input

The first line of input determines the data type to be stored in the list. It can be one of the following two strings:

```
numbers
```

or:

```
strings
```

The input that follows this line has the same syntax and meaning as in *Assignment 3*. The only exception is that the value ***v*** can be a string (i.e., any sequence of characters, numbers, or symbols, but no spaces) if *strings* had been provided at the first line of input.

The following table summerizes the input structure:

input	description
1 ***i v***	add node at index i, with value v
2 ***i***	remove node at index i
3 ***v***	remove first encountered node with value v
0	quit

The program ends if the input is **zero** (regardless of the used data type).

3.3 Output

- For each input (except the last **zero**), the program prints out the following two lines:
  - The first line shows the list's common value (GCD for numbers, or LCS for strings). For example:

    ```
    List's common value: BBB
    ```

  - The second line shows the contents of the list (separated by single spaces) printed in the same order of input (the first number entered is on the left). An example:

    ```
    List contents: BBB DDCA AAC
    ```

- If the list is empty (for example, after removing all the nodes), show the following for numeric lists:

  ```
  List's common value: 0
  List contents:
  ```

  Or, if strings are used: (empty string istead of **0** in the first line)

  ```
  List's common value:
  List contents:
  ```

- On exit (when receiving the value **0**), there is no output.

3.4 General notes

- Only implementations based on *Dynamic Memory Allocation* and *template classes* will receive full points.
- implement *Node* as a template class with *constructor* and *destructor* methods.
- implement *Linkedlist* as a template class with *constructor* and *destructor* methods.
- You may use any algorithm you prefer to calculate the GCD of the list.
- It is recommended to use a *dynamic programming* algorithm (similar to the one implemented here) to find the LCS. Note that the code shown here returns the length (not the actual string) of the LCS.
- The meaning and use of the index ***i*** is the same as in *Assignment 3*.
- Managing *Dynamic Memory Allocation* properly (15% of this assignment's grade) requires avoiding issues like memory leaks. Depending on your OS and/or IDE, there are some tools that can help identify some of these issues (like cppcheck, available on **code01**).
- The use of templates allows the same general class to be instantiated with different types (provided on instantiation). The code snippet below shows how the **Node** can be re-written to achieve this:

```
template<class T>
class Node {
public:
        T value;
        Node *next;

        Node() { // constructor
                next = nullptr;
        }
};
```

  This class can later be instantiated as follows:

```
        Node<int> *int_list;
        Node<float> *float_list;
```

  More details can be found in the slides/links on the class's website (slides 25, 26 of "Class in C++" are a good starting point).

- This general class can have a method (say ***get_common_value()***) that, based on the data type being used, computes and returns the required common value (i.e., GCD in case of numbers, or LCS for strings).

  There are few alternative approaches to implement this. The following are some examples:

  - *Overloading* this method (requires providing parameters with different types) as follows:

```
        long get_common_value(long dummy) {
        ...
        }

        string get_common_value(string dummy) {
        ...
        }
```

- Checking for the type used (using something like *typeid().name()*), to determine the appropriate method to call (this is platform dependent!).
- Defining a top level parent class, and two "specialized" child classes (one for strings, and the other for numbers) inheriting from it. the child classes can then *Override* the base method from the parent with their own type-specific versions (this will be discussed in greater details in coming lectures).

4 Sample input/output

In the following examples we will focus more on handling string lists, since number lists are handled largely in the same way as in *Assignment 3*.

4.1 Sample 1

Input

```
strings
1 0 BBB
1 0 B
1 2 A
2 3
2 2
1 5 BBA
3 BBA
3 B
1 5 ABBBBB
3 BBB
1 1 BBAA
0
```

Output

```
List's common value: BBB
List contents: BBB
List's common value: B
List contents: B BBB
List's common value:
List contents: B BBB A
List's common value:
List contents: B BBB A
List's common value: B
List contents: B BBB
```

```
List’s common value: B
List contents: B BBB BBA
List’s common value: B
List contents: B BBB
List’s common value: BBB
List contents: BBB
List’s common value: BBB
List contents: BBB ABBBBB
List’s common value: ABBBBB
List contents: ABBBBB
List’s common value: BB
List contents: ABBBBB BBAA
```

4.2 Sample 2

Input

```
strings
1 0 BBB
1 0 BBB
1 2 BBAA
3 BBB
2 0
1 5 AAABBB
1 3 BBAAA
2 0
0
```

Output

```
List’s common value: BBB
List contents: BBB
List’s common value: BBB
List contents: BBB BBB
List’s common value: BB
List contents: BBB BBB BBAA
List’s common value: BB
List contents: BBB BBAA
List’s common value: BBAA
List contents: BBAA
List’s common value: BB
List contents: BBAA AAABBB
List’s common value: BB
List contents: BBAA AAABBB BBAAA
List’s common value: AAA
List contents: AAABBB BBAAA
```

4.3 Sample 3

Input

```
strings
1 0 AAABBB
1 0 A
1 2 BBAABB
1 1 AAA
1 4 BBBB
3 A
3 AABBBAAA
1 7 BB
3 A
2 8
3 AAA
1 0 A
2 0
0
```

Output

```
List’s common value: AAABBB
List contents: AAABBB
List’s common value: A
List contents: A AAABBB
List’s common value: A
List contents: A AAABBB BBAABB
List’s common value: A
List contents: A AAA AAABBB BBAABB
List’s common value:
List contents: A AAA AAABBB BBAABB BBBB
List’s common value:
List contents: AAA AAABBB BBAABB BBBB
List’s common value:
List contents: AAA AAABBB BBAABB BBBB
List’s common value:
List contents: AAA AAABBB BBAABB BBBB BB
List’s common value:
List contents: AAA AAABBB BBAABB BBBB BB
List’s common value:
List contents: AAA AAABBB BBAABB BBBB BB
List’s common value: BB
List contents: AAABBB BBAABB BBBB BB
List’s common value:
List contents: A AAABBB BBAABB BBBB BB
List’s common value: BB
List contents: AAABBB BBAABB BBBB BB
```

4.4 Sample 4

Input

```
strings
1 0 AA
2 0
1 2 AAB
```

```
1 0 BBBBA
2 2
1 2 AAABB
1 4 AABBB
3 AAB
2 1
0
```

Output

```
List’s common value: AA
List contents: AA
List’s common value:
List contents:
List’s common value:
List contents:
List’s common value: BBBBA
List contents: BBBBA
List’s common value: BBBBA
List contents: BBBBA
List’s common value: BB
List contents: BBBBA AAABB
List’s common value: BB
List contents: BBBBA AAABB AABBB
List’s common value: BB
List contents: BBBBA AAABB AABBB
List’s common value: BBB
List contents: BBBBA AABBB
```

4.5 Sample 5

Input

```
numbers
1 0 8
1 0 16
1 1 12
3 8
1 0 32
1 1 20
2 1
1 1 2
0
```

Output

```
List’s common value: 8
List contents: 8
List’s common value: 8
List contents: 16 8
List’s common value: 4
```

```
List contents: 16 12 8
List’s common value: 4
List contents: 16 12
List’s common value: 4
List contents: 32 16 12
List’s common value: 4
List contents: 32 20 16 12
List’s common value: 4
List contents: 32 16 12
List’s common value: 2
List contents: 32 2 16 12
```

5 Rubric

Criterion	Possible points	Excellent (max. points)	Satisfactory (partial points)	Unsatisfactory (no points)
Delivery	5 %	on time, using correct file name (see top)		wrong file name
Compiles	5 %	compiles on code01.fit.edu with no errors		does not compile
Runs	5 %	runs on code01.fit.edu with no run-time errors (missing files,..etc)		does not execute
Using template class	25%	all templates (for Node and Linkedlist) implemented correctly	some templates implemented	no templates used
Dynamic Memory allocation	20%	implemented using pointers/Dynamic Memory allocation		implemented using other memory access methods like arrays
Avoiding memory leaks	15%	freeing memory correctly for all pointers & dynamic memory allocation operations		code has memory leaks
Test cases	25%	passes all test cases (correct output in the correct format)	passes some test cases	fails all test cases, or has an endless loop.