CSE 241 Programming Assignment 3

DUE

May 1, 2025, 23:55

Description

- This is an individual assignment. Please do not collaborate
- If you think that this document does not clearly describes the assignment, ask questions before its too late.

This assignment is about implementing and testing classes which use **dynamic memory**. You are going to implement an "advanced" string class.

Design and implement the following class. You can define other classes if you want.(Feel free to add other definitions or members if you need to):

- `WordString`:
  - Stores string data.
  - Can return, replace, remove words in the string.
  - Can add new words before and after any word in the string.
  - Keeps the count of total number of words.
  - Can strip(remove excessive spaces) the string.
  - WordString object supports major string operations such as `=`, `<<`, `>>`, `+`, `==`, etc.

Definition of a `word`

- A word is a string of chars which does not include any spaces(`' '`).
- You can store a word in a `std::string`.
- You can store a word in a `c-string`.
- There is no size limit. A word can be "very" long. Don't assume a `MAX` size.
- There is no count limit. There may be millions of words in a string. (There may be a practical limit depending on your approach but you shouldn't assume a predefined `MAX` number of words.)

The Organization

Approach 1

- You can use `std::string` as member data of `WordString`. The whole `raw` data can be stored in a std:string.
- In addition to the raw data, you have to store the indices and lengths of each word in the string. This storage **HAS TO BE DYNAMICALLY ALLOCATED**. You cannot use `std::vector`.
- If the string changes, you may need recalculate and update the indices of the words.
- Reaching to a word in a very long sentence should not require a search. You should use indices and reach to the data in constant time. If you repeatedly try to find the same words again and again. Unless the data is changed, you have to use indices.

Approach 2

- You can store individual words in a dynamically created array of `std::string` objects. You **HAVE TO USE DYNAMIC ALLOCATION** to store string objects. (You don't know the number of words initially). You cannot use `std::vector`.
- Since you store individual words, you don't have to keep indices. But, you have to remember spaces in between the words. You can create strings which store spaces. You can place these objects in the array of words.
- Instead of creating strings for spaces, you can also decide to keep another dynamic array of integers which store the number of spaces after each word.

Approach 3

- Don't use `std::string` at al. Use a dynamic array or arrays of `c-string`.

- The rest will be similar to Approach 1 or Approach 2.
- This one will be the most efficient one.

Requirements for Dynamically Allocated Data

- Since `WordString` objects point to dynamically allocated data, you have to properly handle the following:
  - Assignment Operator
  - Copy Constructor
  - Destructor
- If you avoid any of these, you will get `0`. If you don't properly handle any of these, your program may crash or produce wrong results.

The Implementation

- After deciding on how to store and organize the data, you should start with the functions about converting string to WordString and vice versa. These operations may be trivial depending on your approach. For example, if you follow Approach 1, you don't have to do anything. You just need a simple function which returns the raw data.

You need to extract individual words and count their lengths(tokenizing). Stream iterators may help here but you cannot repeatedly use them in order to get to the words. If the data is not modified, searching or iterating through the data again and again to find individual words is strictly forbidden. You will get `0`.

Once you figure out individual words, you should use this information. If the data is entirely modified, you may re-run you internal function to find words.

Inserting and deleting individual words will not require finding all the words again. You can shift the indices.

You can use string library functions. For example, inserting a new word can be done with string library functions.

Some of the functions you will implement may use the other functions. For example, operator+() can have a loop inside and call insertAfter() function.

Try not to repeat the functionality. Try to use delegation.

main.cpp

Here is a comprehensive main.cpp example which shows some operations. This example is provided in order to show some of the operations. Write your own `main.cpp` for the actual tests.

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

#include "WordString.h"

int main(){

    WordString ws1, ws2;
    //A WordString object can be initialized with a c-string;
    WordString ws3("This is a c-string literal");
    //A WordString object can be initialized with std::string object
    string boringString = "Just another literal living in this main function";
    WordString ws4(boringString);
    //Lets print some string data
    //This should print the following without quotes
    //"Just another literal living in this main function"
    cout<<ws4<<endl;
    //As you can see, the data is exactly the same with the std::string content.
    //So, WordString can be used like a std::string
```

```
//Return the std::string form of the data
string catcher = ws3.toString();
cout<<catcher<<endl;
//"This is a c-string literal" is printed.
ws3 = boringString;
//ws3 has the same contents with boringString.
//We can even get data from stdin
cin>>ws3;
//Of course, the capture will stop with the first whitespace.
//You don't have to capture a sentence of words.

//WordString specific functionality
//return the number of words.
int c = ws4.numberOfWords();
cout<<c<<endl;
//This prints 8

//Returns one of the words as a copy (index starts from 0)
string word5 = ws4.get(5);
cout<<word5;
//prints "this" without quotes

//replace a word with a new one
string new_word = "new"
ws4.replace(3,new_word);
//replaces "living" with "new"
cout<<ws4.get(3);
//prints "new" without quotes
//The rest of the data is not modified.
cout<<ws4;
//This prints the following
//"Just another literal new in this main function"

//inserting new string. BE CAREFUL, INSERTED STRING MAY INCLUDE MULTIPLE WORDS
//add ONE space before and after the inserted string.
ws4.insertAfter(3,string("inserted string"))
cout<<ws4;
//This prints the following
//"Just another literal new inserted string in this main function"

//remove one word
//Remove one space before or after the removed word.
//Remove the space after if the removed word is the first word.
//Remove a space before, if it is not the first word.
//Do not remove any spaces it there is only one word in the object.
ws4.remove(4);
cout<<ws4;
//Prints: "Just another literal new string in this main function"

//Strip extra spaces.
//Delete spaces if here are multiple concurrent spaces.
//Example:
//word1     word2  word3
wordString ws5 = "word1     word2  word3";
ws5.strip();
//Stripped example:
```

```
    //word1 word2 word3
    cout<<ws5;
    //Prints: "word1 word2 word3"

    //Adding two objects. Insert a space before the second operant.
    ws2 = ws4 + ws3;
    //Adding wordString and a string
    string additional_string("additional data");
    ws2 = ws4 + additional_string;
}
```

Remarks

- Don't forget to delete any dynamic allocation which is not needed anymore.
- The program does not print any error messages.
- You cannot use `std::vector` or a similar implementation.
- Write comments in your code.
- If your code does not compile you will get `0`.
- Do not share your code with your classmates.
- **Remove any print statements which you use for debug purposes.**

Turn in:

- You are going to create a **zip** archive which includes the following files:
  - WordString.h
  - WordString.cpp
- Name of the file should be in this format: **`<full_name>_PA3.zip`**. If you do not follow this naming convention you will loose `-10` points.
- The archive type should be **zip**. The archive should be flat. When extracted, the files **should not** be placed in a subdirectory.
- You don't need to use an IDE for this assignment. Your code will be compiled and run in a command window.
- Your code will be compiled and tested on a Linux machine(Ubuntu). GCC will be used.
- A script may be used in order to check the correctness of your results. So, be careful not to violate the expected output format.
- Provide comments unless you are not interested in partial credit. (If I cannot easily understand your design, you may loose points.)
- You may not get full credit if your implementation contradicts with the statements in this document.

Late Submission

- Not accepted

Grading (Tentative)

- `Max Grade : 100`.
- Multiple tests will be performed.

All of the followings are possible deductions from `Max Grade`.

- Do **NOT** use hard-coded values. If you use you will loose `10pts`.
- No submission: `-100`.
- Compile errors: `-100`.

- Irrelevant code: `-100`.
- Major parts are missing: `-100`.
- Unnecessarily long code: `-30`.
- Inefficient implementation: `-20`.
- Using language elements and libraries which are not allowed: `-100`.
- Not caring about the structure and efficiency: `-30`. (avoid using hard-coded values, avoid hard-to-follow expressions, avoid code repetition, avoid unnecessary loops).
- Significant number of compiler warnings: `-10`.
- Not commented enough: `-10`. (Comments are in English. Turkish comments are not accepted).
- Source code encoding is not `UTF-8` and characters are not properly displayed: `-5`. (You can use 'Visual Studio Code', 'Sublime Text', 'Atom' etc... Check the character encoding of your text editor and set it to `UTF-8`).
- Missing or wrong output values: `Fails the test`.
- Output format is wrong: `-30`.
- Not deallocating memory: `-10`.
- Using `std::vector`: `-100`.
- Does not work with long string: `-50`.
- Infinite loop: `Fails the test`.
- Segmentation fault: `Fails the test`.
- Fails 5 or more random tests: `-100`.
- Fails the test: `deduction up to 20`.
- Prints anything extra: `-30`.
- Unwanted chars and spaces in output: `-30`.
- Submission includes files other than the expected: `-10`.
- Submission does not follow the file naming convention: `-10`.
- Sharing or inheriting code: `-200`.