# CS 162 Computer Science II

## Homework Assignment 4 - Linked Lists 1

### Academic Integrity

**You may NOT, under any circumstances, begin a programming assignment by looking for completed code on StackOverflow or Chegg or any such website, which you can claim as your own. Please check out the Student Code of Conduct at PCC.**

The only way to learn to code is to do it yourself. The assignments will be built from examples during the lectures, so ask for clarification during class if something seems confusing. If you start with code from another source and just change the variable names or other content to make it look original, you will receive a zero on the assignment.

I may ask you to explain your assignment verbally. If you cannot satisfactorily explain what your code does, and answer questions about why you wrote it in a particular way, then you should also expect a zero.

In this assignment you will create a list of word frequencies for an input file.

# CS 162 Computer Science II

## Homework Assignment 4 - Linked Lists 1

## Purpose

The focus of this assignment is working with dynamic linked lists and dynamic arrays.

After completing this assignment you will be able to:

- Write code using classes and objects that use dynamic linked lists and arrays
- Use dynamic C-strings
- Use dynamic single link (forward) linked lists
- Insert, unlink, and search for elements in a linked list
- Avoid memory errors such as seg fault and memory leaks
- Organize source code in multiple header and implementation files (.h and .cpp files).
- Implement and use accessors and mutators for a class
- Read from input data files

## Overview

This Homework Assignment includes zyBook exercises and a programming assignment. Please read all of the instructions carefully - there are a lot of details that you need to know!

This assignment is a re-implementation of Homework 3, except the word index will be a dynamic linked list instead of a dynamic array. You should continue to use right-sized dynamic c-strings - only the list of `word` objects is being transformed into a linked list.

There is no limit to the number of words in the input file, nor is there any limit on the number of characters in a word. Also keep in mind that the input file may contain blank lines or lines with only whitespace characters.

In this assignment you will create and submit eight (8) files to D2L, namely

1. `algorithm.pdf` - An algorithm design document exported to PDF format
2. `output.txt` - A log of your program's execution on the Linux server
3. `list.h` - A C++ header file that contains the declarations related to the `list` class
4. `word.h` - A C++ header file that contains the declarations related to the `word` class
5. `main.h` - A C++ header file that contains the declarations related to the `main` module
6. `list.cpp` - A C++ implementation file that contains the implementation of `list.h`
7. `word.cpp` - A C++ implementation file that contains the implementation of `word.h`
8. `main.cpp` - A C++ implementation file that contains the implementation of `main.h`

**You should use these file names** for all submissions, and your submission is not complete unless all files are submitted to D2L.

## zyBook Exercises

You should complete the following zyLabs:

1. zyLab 21.10 - Inventory. This lab introduced the operations of a dynamic linked list.
   .

Once you have completed this zyLab with a score 100% you will need to create a screenshot image of your completion statistics from zyBook, and include that in your algorithm document for this assignment. Note that “successfully completed” means that you received 100% on both labs.

*The zyLabs a required element of this assignment* - your submission is not complete unless the success statistics are included in your algorithm document.

## Programming Task

### Introduction

The goal of this assignment is to create a list of word frequencies for an input file. The list must be ordered by ASCII code (which is how the `strcmp` function orders strings), must not contain any duplicate words, and must be implemented as a right sized array.

For example, suppose the input file is:

```
> cat input.txt
hello world this is the first line
hello world this is the second line
hello world this is the third line
goodbye world
>
```

The word frequency list would be something like:

```
> main
Enter input file name: input.txt
File "input.txt" has 10 words in 5 lines:
Word: "first" Count: 1
Word: "goodbye" Count: 1
Word: "hello" Count: 3
Word: "is" Count: 3
Word: "line" Count: 3
Word: "second" Count: 1
Word: "the" Count: 3
Word: "third" Count: 1
Word: "this" Count: 3
Word: "world" Count: 4
```

The list shown above from my sample solution displays an ordered list of words and number of times each word appears in the file (words in quotes, ASCII order) with no duplicate words. The output also includes the total number of words and lines in the input file, but that is extra, and you do not have to display that information in your program.

This assignment uses the notion of a `list` variable. In earlier assignments a list was merely an array of items, and we now extend this to the idea that a list is a class that includes both the elements of the list as well as operations on those elements.

In this homework assignment your list class will maintain a dynamic linked list, instead of a dynamic array.

Using C++ jargon we are introducing you to the following ideas:

1. A *collection* is a group of elements. An array is a collection, as is a linked list. A collection represents just a pile of data. Operations on the data (insert, remove, find, et. al.) are **not** components of a collection.

2. A *container* is an object that holds and manages a collection. In other words, a container is created when a collection is combined with all of the operations on the data.

> The container contains the collection, and also the insert, remove, find, and other operations that use the collection.
>
> The C++ `vector` and `string` data types are containers.

In C++ the data type of the underlying collection usually is represented by a separate class (think of an array or `vector` of C++ `string` objects). We'll follow that practice in this assignment by implementing a container class named `list` that manages a collection of objects of type `word`. The `word` class will represent the data for a single word in the file (e.g. one line of the display), and the `list` class will represent the entire index.

Container classes in C++ typically are implemented using *class templates*, which is a technique not discussed in CS 162. Do not use templates in this assignment, no matter what stackoverflow.com might try to tell you.

## Classes

`word` Class

The `word` class is the basic data element of the collection, and contains the following private data members. You must use the names and data types shown below for your private data:

```
private:
     char *data;
     int count;
```

`data` is a dynamic c-string that contains the word text. `count` is the number of times the word appears in the input file.

The `word` class must include a default constructor, destructor, copy constructor, and copy assignment operator. You may include other constructors, accessors, mutators, and helper functions as you desire. Do not include any operator overloads except the copy assignment operator.

Note that in this homework assignment the `word` class is identical to that used in Homework 3 - no changes should be required.

`list` Class

The `list` class is a container class that maintains a list of `word` objects, and this class is different from the `list` class in Homework 3 because it manages a linked list, not an array.

The `list` class contains the following private data members. You must use the names and data types shown below for your private data:

```
private:
      struct node {
            word * data;
            node * next;
      };
      node * index;
```

Linked list pointers must be available in each element of the linked list, and a common way to provide linked list pointers for objects is to “wrap” them in a simple struct. This is the technique used in Lab 21.10, and is fine for this assignment. A separate module for the struct is NOT required, but note that the struct uses dynamic fields, so you’ll need to be careful with your memory management.

`index` is the head pointer for a linked list of `node` elements (which contain the word objects). array of `word` objects. As before, the list should be an ordered list of no duplicates, i.e. no matter how many times a word appears in the input file, the word should appear only one time in the `index` array. Your `list` class should not contain any other data elements.

The `list` class must include a default constructor, destructor, copy constructor, and copy assignment operator. You may include other constructors, accessors, mutators, and helper functions as you desire. Do not include any operator overloads except the copy assignment operator.

You will find it helpful to include `insert()` and a `print()` methods in your `list` class to make it easy to insert words into the `index` array, and to make it easy to display the contents of the `index` array.

## `main` Module

The `main()` function should perform the following tasks. This is not intended as a design document, merely an outline of the logic expected in your `main()` function.

1. Declare and initialize a variable of type `list`.

2. Prompt the user and accept an input file name. If the input file name cannot be opened, reprompt the user until a valid input file name is entered. The input file name can be of any length, and can contain any characters except a newline (‘\n’).

3. Read each word in the input file. Ensure each word is inserted into the `list` variable. Add 1 to the count for the word every time the word appears in the input file.

4. Close the input file.

5. Display the list of words, with each word and its count on one line.

## Additional Requirements For This Assignment

1. You may use features from the following libraries:
    a. <iostream>
    b. <fstream>
    c. <cstring>
    d. <cctype>

    You may not use any C++ strings, algorithm classes, container classes, or memory management classes (or features of these classes).

    Let's face it - what I am asking you to do in this assignment can be done much more easily if you were allowed to use all of the features of C++. But you are **not** allowed - the goal of this assignment is to practice with dynamic memory - so you get to do things "the hard way".

2. The list of words must be a single linked list, using forward links. The linked list must be "right sized", i.e. the head pointer begins as nullptr, and elements are added as needed.

3. All arrays must be dynamic. Do not use any static arrays. All c-strings must be implemented in dynamic char arrays.

4. All arrays must be "right sized" at all times. There should never be any unused elements in any array. This means the array pointer must be initialized to `nullptr`, and new memory is allocated only when a new element is added to the array.

5. Array sizes should increase by 1 when a new element is inserted, and decrease by 1 when an element is removed.

6. All pointers must be initialized to `nullptr`. Arrays with no data should not have any allocated memory.

## Programming Notes

1. The term "word" in this assignment has a specific technical meaning, namely a sequence of non-whitespace characters delimited by whitespace. A single operator or brace may be a word - the term has nothing to do with English words, or words in any particular language.

   Whitespace characters can be identified by using the `isspace()` function from `<cctype>`.

2. Reading input into a dynamic c-string requires a very different technique than you have used with static c-strings. In order to satisfy the requirement for a "right sized" array you must read each character individually, expand the `char` array, and then add the character to the `char` array.

   There are no standard C++ functions that are compatible with dynamic c-strings, so you'll need to implement your own input logic. The basic algorithm for reading a string into a dynamic char array is shown below. Please note that this isn't intended as executable code - all I'm trying to do is show the logic:

```
char * buffer = nullptr:
int size = 0;
char byte = 0;
while(in.good() == true && in.peek() != EOF) {
     byte = in.get();
     If (buffer == nullptr) {
         buffer = new char[size + 2];
         memset(buffer,0,size+2);
```

```
            Buffer[0] = byte;
            Size = 1;
         } else {
            char * temp = new char[size + 2];
            memset(buffer,0,size + 2);
            strcpy(temp,buffer);
            delete [] buffer;
            buffer = temp;
            buffer[size++] = byte;
         }
      }
```

3. The `strtok()` function from `<cstring>` can be used to parse a line of input into individual words. `strtok()` is mentioned but not explained in the course material, so here is a code snippet showing its use:

```
> cat main.cpp
#include <iostream>
#include <cstring>
using namespace std;

int main() {
     char string[] = "this is a line of text";
     char * token = strtok(string," ");
     while(token != nullptr) {
          cout << "Word: " << token << endl;
          token = strtok(nullptr, " ");
     }
     return 0;
}
```

The output from the program is:

```
> a.out
Word: this
Word: is
Word: a
Word: line
Word: of
Word: text
>
```

In the **first** call to `strtok` the first parameter is the c-string to parse, and the second parameter is a c-string of delimiter characters (i.e. a string of all whitespace characters). The char pointer

`token` is set to the first word of the string. In the `while` loop the first parameter to `strtok` is `nullptr`, indicating that parsing should continue using the original c-string. The `token` pointer is updated to the next word in the line. `strtok` returns `nullptr` when there are no more words in the line, and this is used as the termination condition for the `while` loop.

## General Requirements For Classes

As you create classes for this assignment, please keep in mind that there are some members that you **must** implement in every class. You may implement additional members as needed, but all classes must:

1. Implement a *default constructor* that initializes the object to an "empty" or "no data" state. The object is said to be in its "default state" when the default constructor is done.

   The "no data" state for pointers is to have a value of `nullptr`. All pointers must be set to `nullptr` by the default constructor.

2. Implement a *copy constructor* that initializes the object to be identical to a source object. The copy constructor must make a copy of all dynamic data, so that each object has its own private copy of the data. The original source object is not changed in any way.

3. Implement a *destructor* that resets the object to its default state, i.e. the same state as is set by the default constructor. The destructor is responsible for ensuring that all dynamic memory used by the object is released, so typically the destructor will include `delete` statements.

4. Implement a *copy assignment* operator that assigns the object to be identical to a source object. The copy assignment operator must make a copy of all dynamic data, so that each object has its own private copy of the data. Note that in order to prevent memory leaks the copy assignment operator must delete any existing dynamic memory before creating new dynamic memory for the copies.

5. Use only `private` data - all variables and constants must be `private` in the class. This may mean that accessors and mutators are required as well.

In addition to the "must do" items listed above, there are some "never do" items that are forbidden in CS 162:

1. Do not use `friend`. This is not part of CS 162.

2. Do not create class hierarchies or implement inheritance or virtual functions. This is not part of CS 162.

3. Do not use lambda functions in this assignment. These are discussed in CS 162, but you won't get to use them in assignments.

4. Do not use static class members. Again, this is not a CS 162 topic.

5. Do not use template classes or template functions. Yet again, this is not a CS 162 topic.

## General Requirements for Dynamic Memory

Any program written in CS 162 that uses dynamic memory should satisfy the following requirements:

1. Do not use smart pointers, helper classes, or any features from the `<memory>` library. Use "naked" pointers only.

2. Dynamic memory must be allocated and released using C++ `new` and `delete` operators. Do not use `malloc()`, `calloc()`, `realloc()`, `free()`, or any other C specific memory management.

3. Two different objects should not share dynamic data, i.e. two different objects should not both have pointers to the same dynamic data. This affects accessor design when private data is dynamic, since a **copy** of the dynamic data must be returned by the accessor, rather than having the accessor provide access to the object's private data.

4. All dynamic variables must be initialized when they are allocated, just as all static variables must be initialized when they are declared

5. All dynamic memory allocated by your code must be released by your code before the `main()` function ends. SImply put, memory leaks are bad (for your grade).

6. Your program should not reference any unallocated memory. Do not reference memory blocks after they have been released, and do not reference addresses outside of allocated memory blocks.

7. Your code must not have any memory errors when executed. Use `valgrind` or a similar profile tool to verify that your program does not have any memory errors.

# Submission

1. Complete the zyBook exercises
2. Open the Algorithmic Design Document, make a copy, and follow the steps to create your algorithm.

3. Export your algorithm document to a PDF file named `algorithm.pdf`
4. Create your source code files, build and test your program.
5. All code must follow the CS 162 C++ Style Guide guidelines.
6. Transfer your source code to the PCC Linux Server
7. Compile and execute your code on the PCC Linux server
    a. Type "`script output.txt`" on the command line and it will start recording your session in a file called "`output.txt.`"
    b. Run your program
    c. Type "`exit`" to stop recording.
8. Upload your source code files, algorithm document, and output log file to D2L. Please upload them as individual files - do not zip or tar them into a single archive file.

## Testing, Sample Run, and Success Criteria

Your program should produce correct output, with no memory errors, for any input text file. There is no limit to the number of lines, blank lines, words, or word length in the input - your program should use dynamic arrays and be able to accommodate input of any size.

Be sure to test your program using `valgrind` or a similar program to ensure there are no memory errors. This includes memory leaks, invalid references, segmentation faults, and any other memory error. Your program must perform correctly on the PCC Linux server.

Test your program with a variety of different input files. You should test with:

1. Each of your source code files (.h and .cpp). These files usually have a good variety of words and blank lines.

2. One or more of the following large files. **Do not make copies of these files on the server** - you do not need a copy and you do not want to waste Linux disk space and incur the wrath of the System Administrator.
    a. `/home/courses/doug.jones/162/milton_paradise_lost.txt`. The full text of the epic poem by John Milton. "Awake, arise or be forever fall'n."

    b. `/home/courses/doug.jones/162/homer_iliad.txt`. The full text of the epic poem by Homer. "We men are wretched things."

    c. `/home/courses/doug.jones/162/homer_odyssey.txt`. The full text of the epic poem by Homer. "Each man delights in the work that suits him best."

Here is a sample run with a short input file. Your program does not need to display the total number of words or lines in the file - I added that for another purpose and I'm too lazy to fix it for this assignment sample run!

```
> cat input.txt
hello world this is the first line
hello world this is the second line
hello world this is the third line
goodbye world
> main
Enter input file name: input.txt
File "input.txt" has 10 words in 5 lines:
Word: "first" Count: 1
Word: "goodbye" Count: 1
Word: "hello" Count: 3
Word: "is" Count: 3
Word: "line" Count: 3
Word: "second" Count: 1
Word: "the" Count: 3
Word: "third" Count: 1
Word: "this" Count: 3
Word: "world" Count: 4
>
```

## General Requirements for All Homework Assignments

- ❏ Code may be written using any development environment, but must use Standard C++ and must be successfully tested on the PCC Linux server. Code usually is graded on Linux, so if your program doesn't work on Linux then it doesn't work.

- ❏ PSU-bound students are strongly encouraged to do all development on the PCC Linux server.

- ❏ Open the Algorithmic Design Document, make a copy, and follow the steps to create your algorithm. Be sure to include your screenshot of the zyBooks completion in the algorithm document.

  Download the algorithm document as a PDF file, and submit the PDF file to D2L as part of this assignment.

- ❏ You must express your algorithm as **pseudocode. Do not use a flowchart in CS 162**. Please note that your pseudocode must follow the syntax requirements shown in the document - you may **not** use C++ as a substitute for pseudocode.

## Additional Support

- ❏ Post a question for the instructor in the Ask Questions! area of the Course Lobby.