CptS**122 – Data Structures                                                                                             **

**Programming Assignment 1: Analyzing Fitbit Data**

**Assigned:** Friday, January 22, 2021

**Due:**Wednesday, February 3, 2021 by midnight

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Apply basic techniques for data cleansing or data cleaning

*      Analyze a medium set of data and compute results

*      Filter a dataset

*      Implement deduping methods

*      Compose C language programs

*      Create and compile a program using Microsoft Visual Studio 2019

*      Execute a program in Microsoft Visual Studio 2019

*      Create basic test cases for a program

*      Apply arrays, strings, and pointers

*      Define and apply structs

*      Parse strings from files

*      Convert strings to numerical values

*      Summarize differences between array notation and pointer notation

*      Apply basic string handling library functions

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Access Microsoft Visual Studio 2019 Integrated Development Environment (IDE)

*      Design and develop a small or medium sized program in any language

*      Apply sequential, conditional, and iterative constructs

*      Design and implement user-defined functions

**III. Overview & Requirements:**

The intent of this assignment is to review concepts from your prior “CS 1” course and to challenge and enhance those concepts.

Fitbit is a company that builds wearable technology devices that track various activities. The devices have sensors that measure number of steps and distance walked, heart rate, sleep quality, floors climbed, and calories burned. In this assignment, you will analyze data that was generated from a Fitbit devices. The data is stored in a comma-separated values (.csv) file that you will find at: <https://eecs.wsu.edu/~aofallon/cpts122/progassignments/FitbitData.csv>. The entries in this file were merged from two different devices. You will need to *filter* any data that is not related to the target patient. The first data entry in the file contains the target. You will also need to *dedupe*any entries that appear multiple times and perform *data cleansing* any entries that have missing fields. A .csv file stores data as plaintext in tabular form. Each row in the file is considered a *record*. Each record consists of *fields* separated by commas.

In particular, you will analyze 24 hours of data. Each record in the “FitbitData.csv” represents one minute of data and consists of eight fields. These include the following:

1.    Patient ID

2.    Minute

3.    Calories

4.    Distance (in miles)

5.    Floors

6.    Heartrate

7.    Steps

8.    Sleep level

*What data structures are required?*

In this assignment, you must define a C struct to store a subset of the Fitbit data fields as follows:

typedef struct fitbit

{

char patient[10];

     char minute[9];

     double calories;

     double distance;

     unsigned int floors;

     unsigned int heartRate;

unsigned int steps;

Sleep sleepLevel;

} FitbitData;

The type Sleep is enumerated and must be defined as follows:

typedef enum sleep

{

     NONE = 0, ASLEEP = 1, AWAKE = 2, REALLYAWAKE = 3

} Sleep;

You must also define an array of FitbitData that can store 24 hours of minute data. Hence, you must declare an array of size 1440. You have the freedom to decide on other data structures and variables that you need for the assignment.

*What are the other requirements?*

This program does not require any user input! However, you will need to display some results to the screen!

-       You must open “FitbitData.csv” for mode read; check for success

-       You must read each record in the file as a *string*, one line at a time; if the record does not belong to the *target* patient, then it should be discarded

-       You must parse each record into the corresponding fields, and store into the FitbitData array; note: not all fields have values, some are “empty” or null; if some of the fields are “empty” or null, then you must perform *data* *cleansing* and insert values to construct a record that is consistent with the others; the data inserted should not represent *valid* values

-       You must compute the total calories burned, distance walked in miles, floors walked, and steps taken

-       You must compute average heartrate over the 24 hour period

-       You must report the max steps taken in one minute over the 24 hour period; if there are multiple minutes throughout the day where the max is discovered, then report the one that is the latest in the 24 hour period

-       You must report the longest consecutive range of poor sleep; a range is defined as one or more consecutive minutes where the sleepLevel > 1; the poorest sleep is not based on the length of the range, but the sum of the sleep levels in the range; the max sum of the ranges is considered the poorest sleep (report the starting and ending minutes of range)

-       You must open “Results.csv” for mode write; this will either create a new .csv or overwrite an existing one

-       You must output two lines to “Results.csv” and to the screen in the following format:

o   Line 1: Total Calories,Total Distance,Total Floors,Total Steps,Avg Heartrate,Max Steps,Sleep

o   Line 2: valueCalories,valueDistance,valueFloors,valueSteps,valueHeartrate,valueMax,valueSleepStart:valueSleepEnd

o   Lines 3 – N: the filtered, deduped, and cleansed data in the original FitbitData.csv format

-       You must close “FitbitData.csv” and “Results.csv”

**IV. Submitting Assignments:**

1.    Using Blackboard Learn <https://learn.wsu.edu/webapps/login/> submit your assignment. You will submit your assignment in the ***lab*** Blackboard space. Under the "Content" link navigate to the "Programming Assignment Submissions" folder and upload your solution to the appropriate “Assignment” space. You must upload your solution, through an attachment, as <your last name>\_pa1.zip by the due date and time.

1. Your project must contain one header file (a .h file), two C source files (which must be .c files), and project workspace.
2. Your project must build properly. The most points an assignment can receive if it does not build properly is 65 out of 100.

**V. Grading Guidelines:**

This assignment is worth 100 points. Your assignment will be evaluated based on a successful compilation and adherence to the program requirements. We will grade according to the following criteria:

*      5 pts for correctly defining struct FitbitData

*      3 pts for correctly defining enum Sleep

*      3 pts for correctly opening “FitbitData.csv” for mode read

*      2 pts for correctly checking the success of the file opening operation

*      10 pts for correctly reading each record in the file as a *string*, one line at a time, and *filtering* and *deduping* any lines that do not belong to the *target* patient or are duplicates

*      15 pts for correctly *cleaning* the data and parsing each record into the corresponding fields

*      5 pts for correctly converting the strings to the field types

*      5 pts for correctly storing the fields into the FitbitData array

*      3 pts for correctly computing the total calories burned

*      3 pts for correctly computing the total distance walked in miles

*      3 pts for correctly computing the total floors walked

*      3 pts for correctly computing the total steps taken

*      4 pts for correctly computing the average heartrate over the 24 hour period

*      8 pts for correctly determining the max steps taken in one minute over the 24 hour period

*      8 pts for correctly determining the longest consecutive range of poor sleep

*      3 pts for correctly opening “Results.csv” for mode write

*      8 pts for correctly formatting and outputting the data, without any duplicate entries, to “Results.csv” and to the screen

*      2 pts for correctly closing the “FitbitData.csv”

*      2 pts for correctly closing the “Results.csv”

*      5 pts for appropriate top-down design of functions and good style

**CptS 122 – Data Structures                                                                         **

**Lab 2: Data Structures and Dynamic Singly Linked Lists in C**

**Assigned:** Week of February 1, 2021

**Due:**At the end of the lab session

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design, implement and test a dynamic singly linked list in C

*      Programmatically traverse through a singly linked list

*      Implement insertions, in order, into a singly linked list

*      Implement deletions from a singly linked list

*      Support modifications and printings of data in a singly linked list

*      Develop a menu system

*      Apply standard library functions malloc () and free ()

*      Design and implement basic manual unit tests

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Compose a small C language program

*      Compile a C program using Microsoft Visual Studio 2019

*      Create test cases for a program

*      Apply and implement arrays and strings in C

*      Apply and implement recursion in C

*      Apply and implement structs in C

*      Apply and implement pointers in C

*      Apply and implement dynamic memory in C

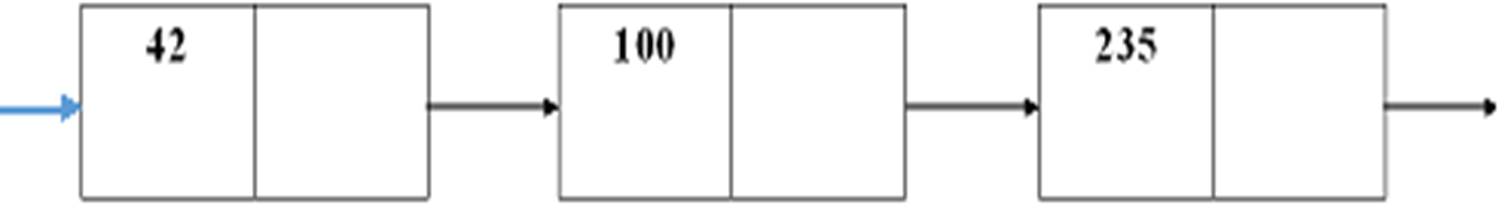
**III. Overview & Requirements:**

This lab, along with your TA, will help you navigate through designing, implementing, and testing a dynamic linked list.

Labs are held in a “closed” environment such that you may ask your TA questions. Please use your TAs knowledge to your advantage. You are required to move at the pace set forth by your TA. You must work in teams assigned by your TA. However, I encourage you to compose your own solution to each problem. Have a great time! Labs are a vital part to your education in CptS 122 so work diligently.

**Tasks:**

1.    Linked lists may be used to implement many real world applications. Recall, linked lists are data structures, which represent collections of nodes that may be accessed sequentially via a pointer to the first node. A node contains data and a pointer to the next node in sequence. When the last node in the list is reached, its next pointer is NULL. A logical view of a singly linked list of integers is illustrated below:



Build an application, in C, for storing contact information (you must have one header file and two source files). For each contact you must store a name, phone number, email, and professional title. Your application must support insertions in order (based on last name), deletions, modifications, and printings of contacts. For this task you will be required to implement a dynamic singly linked list, which grows and shrinks at runtime. Build a menu that allows the user to add, remove, edit, print, store, and load contact information. The *store* feature should write contact information found in the list to a file. The *load* feature should read contact information from the same file into the list. Note: you should store the contact information in a struct called Contact. Each Node must be a struct, which consists of a Contact and a pointer to the next Contact in the list. Please see below.

typedef enum boolean

{

            FALSE, TRUE

} Boolean;

typedef struct contact

{

            char name[50];

            char phone[12]; // 18005557577

            char email[50];

            char title[20];

      } Contact;

typedef struct node

{

            Contact data;

            struct node \*pNext;

} Node;

Which list operations should you support? There are more than the ones listed below!

// Description: Allocates space for a Node on the heap and initializes the Node with the information found in newData.

// Returns: The address of the start of the block of memory on the heap or NULL if no memory was allocated

Node \* makeNode(Contact newData);

// Description: Uses makeNode () to allocate space for a new Node and inserts the new Node into the list in alphabetic order ('a' - 'z')

//              based on the name field

// Returns: TRUE if memory was allocated for a Node; FALSE otherwise

Boolean insertContactInOrder(Node \*\*pList, Contact newData);

// Description: Deletes a Contact in the list based on the name field; deletes the first occurence of the name

// Returns: TRUE if the Contact was found; FALSE otherwise

Boolean deleteContact(Node \*\*pList, Contact searchContact);

// Description: Edits a Contact in the list based on the name field; edits the first occurence of the name

// Returns: TRUE if the Contact was found; FALSE otherwise

Boolean editContact(Node \*pList, Contact searchContact);

// Description: Loads all Contact information from the given file, in alphabetic order, based on the name, into the list

// Returns: TRUE if all Contacts were loaded; FALSE otherwise

Boolean loadContacts(FILE \*infile, Node \*\*pList);

// Description: Stores all Contact information from the list into the given file

// Returns: TRUE if all Contacts were stored; FALSE otherwise

Boolean storeContacts(FILE \*infile, Node \*pList);

// Description: Prints all contact information in the list

// Returns: Nothing

void printList(Node \*pList);

2.    Test your application. In the same project, create one more header file testList.h and source file testList.c (for a total of at least five files). The testList.h file should contain function prototypes for test functions you will use on your list functions. The testList.c source file should contain the implementations for these test functions. You will be designing and implementing *unit* tests. You will have at least one test function per application function. Your test functions must display a message “test failed” or “test passed” depending on the results. For example, you will have an application function called deleteContact() (or a function very similar) that was used to remove contact information from the list. In this task, you will need to create a test function called testDeleteContact() that passes in various contact information directly into deleteNode() to see if it works correctly.

**IV. Submitting Labs:**

*      You are not required to submit your lab solutions, unless you are unable to attend them synchronously. You should keep them in a folder that you may continue to access throughout the semester.

**V. Grading Guidelines:**

*      This lab is worth 10 points. Your lab grade is assigned based on completeness and effort. To receive full credit for the lab you must show up on time, continue to work on the problems until the TA has dismissed you, and complete at least 2/3 of the problems.

**CptS 122 – Data Structures                                                                                             **

**Programming Assignment 2: Digital Music Manager & Doubly Linked Lists – Part I**

**Assigned:** Wednesday, February 3, 2021

**Due:**Friday, February 12, 2021 by midnight (extended date)

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design and implement a dynamic doubly linked list

*      Allocate and de-allocate memory at runtime

*      Manipulate links in a dynamic list

*      Insert items into a dynamic linked list

*      Delete items from a dynamic linked list

*      Edit items in a dynamic linked list

*      Traverse a dynamic linked list

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Compose C language programs

*      Compile a program using Microsoft Visual Studio 2019

*      Create basic test cases for a program

*      Apply arrays, strings, and pointers

*      Summarize differences between array notation and pointer notation

*      Apply pointer arithmetic

*      Apply basic string handling library functions

*      Define and implement structures in C

*      Summarize the operations of a linked list

**III. Overview & Requirements:**

Many of us have large digital music collections that are not always very well organized. It would be nice to have a program that would manipulate our music collection based on attributes such as artist, album title, song title, genre, song length, number times played, and rating. For this assignment you will write a basic digital music manager (DMM).

Your DMM program must have a text-based interface which allows the user to select from a *main* *menu* of options including: (1) load, (2) store, (3) display, (4) insert, (5) delete, (6) edit, (7) sort, (8) rate, (9) play, (10) shuffle, and (11) exit. For Part I of the assignment, you will only need to complete the *main menu*, (1) load, (2) store, (3) display, (6) edit, (8) rate, (9) play, and (11) exit features. The other features will be completed in the next part of the assignment.

  *What must the main menu contain?*

The *main* menu must display the following commands:

(1)   load

(2)   store

(3)   display

(4)   insert

(5)   delete

(6)   edit

(7)   sort

(8)   rate

(9)   play

(10) shuffle

(11) exit

After a command is selected and completed, your program must display the *main* menu again. This procedure will continue until the “exit” command is selected.

  *What must “load” do?*

The “load” command must read all *records* from a file called musicPlayList.csv ([you may find a sample file here](https://eecs.wsu.edu/~aofallon/cpts122/progassignments/musicPlayList.csv)) into a dynamic *doubly* linked list. The doubly linked list is considered the main playlist. As each record is read from the file, it must be inserted at the front of the list. Each *record* consists of the following attributes:

*      Artist – a string

*      Album title – a string

*      Song title – a string

*      Genre – a string

*      Song length - a struct Duration type consisting of seconds and minutes, both integers

*      Number times played – an integer

*      Rating – an integer (1 – 5)

Each attribute, in a single record, will be separated by a comma in the .csv (comma separated values) file. This means that you will need to design an algorithm to extract the required attributes for each record. Each field in each record will have a value. You do not need to check for null or empty values.

You must define a struct called Record to represent the above attributes. Also, do not forget that the *Song Length* must be represented by another struct called Duration. Duration is defined as follows:

*      Minutes – an integer

*      Seconds – an integer

Finally, each struct *Node* in the doubly linked list must be defined as follows:

*      Data – a Record

*      Pointer to the next node

*      Pointer to the previous node

  *What must “store” do?*

The “store” command writes the current *records*, in the dynamic doubly linked list, to the musicPlayList.csv file. The *store* will completely overwrite the previous contents in the file.

  *What must “display” do?*

The “display” command prints records to the screen. This command must support two methods, one of which is selected by the user:

1.    Print all records.

2.    Print all records that match an artist.

  *What must “edit” do?*

The “edit” command must allow the user to find a record in the list by *artist*. If there are multiple records with the same artist, then your program must prompt the user which one to edit. The user may modify all of the attributes in the record.

  *What must “rate” do?*

The “rate” command must allow the user to assign a value of 1 – 5 to a song; 1 is the lowest rating and 5 is the highest rating. The rating will *replace* the previous rating.

  *What must “play” do?*

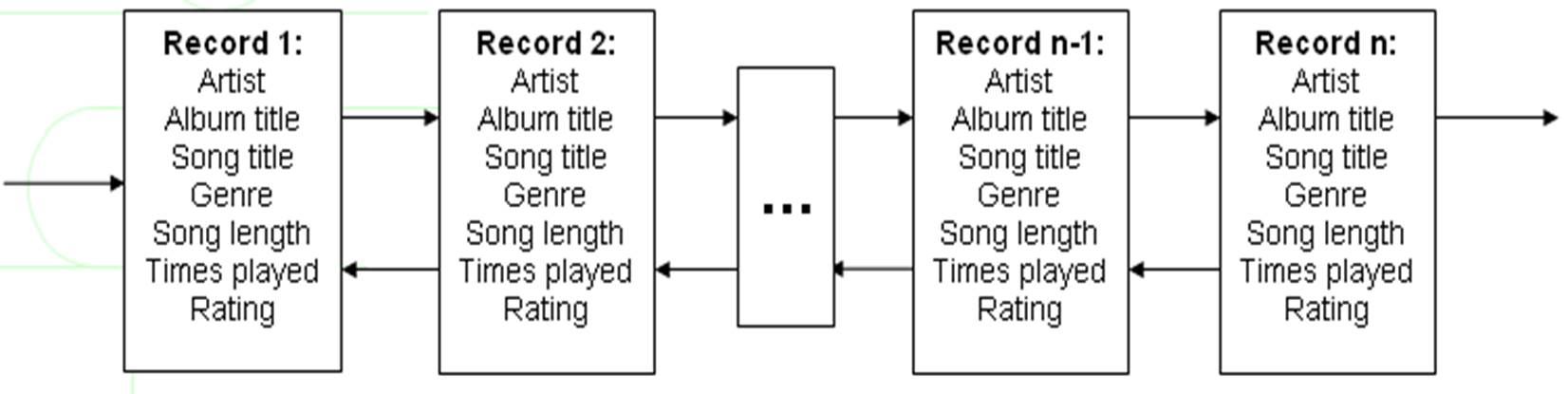
The “play” command must allow the user to select a song, and must start “playing” each song in order from the current song. “Playing” the song for this assignment means displaying the contents of the record that represents the song for a short period of time, clearing the screen and showing the next record in the list, etc. This continues until all songs have been played.

  *What must “exit” do?*

The “exit” command saves the most recent list to the musicPlayList.csv file. This command will completely *overwrite* the previous contents in the file.

**IV. Logical Block Diagram**

The logical block diagram for your doubly linked list should look like the following:



As you can see from the illustration a doubly linked list has a pointer to the next node and the previous node in the list. The first node’s previous node pointer is always NULL and the last node’s next pointer is always NULL. When you insert and delete nodes from a doubly linked list, you must always carefully link the previous and next pointers.

**V. Submitting Assignments:**

1.       Using Blackboard Learn <https://learn.wsu.edu/webapps/login/> submit your assignment. You will submit your assignment in the ***lab*** Blackboard space. Under the "Content" link navigate to the "Programming Assignment Submissions" folder and upload your solution to the appropriate “Assignment” space. You must upload your solution, through an attachment, as <your last name>\_pa2.zip by the due date and time.

1. Your project must contain at least one header file (a .h file), two C source files (which must be .c files), and a local copy of the .csv file.
2. Your project must build properly. The most points an assignment can receive if it does not build properly is 65 out of 100.

**VI. Grading Guidelines:**

This assignment is worth 100 points. Your assignment will be evaluated based on a successful compilation and adherence to the program requirements. We will grade according to the following criteria:

*      5 pts – Appropriate top-down design, style, and commenting according to class standards

*      4 pts – For correct definition of struct Record

*      2 pts – For correct definition of struct Duration

*      3 pts - For correct definition of struct Node

*      5 pts – For correctly displaying the *main* menu, getting the command from the user, and executing the command

*      3 pts – For *looping* back to main menu after a command is executed

*      21 pts – For correctly constructing a *doubly* linked list, including:

1.    (6 pts) For correct implementation a makeNode() function, which allocates space for a struct Node on the heap, and initializes the node

2.    (9 pts) For correct implementation of insertFront() function, which calls makeNode() and returns 1 for successfully allocating space for a node; 0 otherwise

3.    (6 pts) For correct implementation of printList(), which visits each node in the list and prints out the contents of the record

*      15 pts – Correct “load” command implementation

1.    (2 pts) For correctly opening musicPlayList.csv for mode “read”

2.    (6 pts) For correctly extracting each attribute from each record in the file

3.    (5 pts) For correctly using insertFront()

4.    (2 pts) For correctly closing musicPlayList.csv

*      13 pts – Correct “store” command implementation, which writes the records in the list to the musicPlayList.csv file.

1.    (3 pts) For opening musicPlayList.csv for mode “write”.

2.    (10 pts) For correctly writing all the *records* in the list to the file, maintaining the .csv format

*      7 pts – For correct “display” command implementation

1.    (2 pts) For displaying all records by using printList ()

2.    (5 pts) For *searching* for specific records based on *artist* and displaying matching record - should be able to use the same search function as used in the “edit” command

*      7 pts – Correct “edit” command implementation

1.    (2 pts) For *searching* for specific records based on *artist* – should be able to use the same search function as used in the “display” command

2.    (5 pts) For *editing* the record specified by the user

*      3 pts – Correct “rate” command implementation

*      7 pts – Correct “play” command implementation

1.    (2 pts) For playing all songs in order until the end of the list

2.    (5 pts) For *searching* for specific song based on *song title* and playing all songs until the end of the list has been reached

*      5 pts – Correct “exit” command implementation, which writes the records in the list to the musicPlayList.csv file, and exits the program.

**CptS 122 – Data Structures                                                                                             **

**Programming Assignment 3: Digital Music Manager & Doubly Linked Lists – Part II**

**Assigned:** Wednesday, February 10, 2021

**Due:**Monday, February 22, 2021 by midnight (extended date)

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design and implement a dynamic doubly linked list

*      Allocate and de-allocate memory at runtime

*      Manipulate links in a dynamic linked list

*      Insert items into a dynamic linked list

*      Delete items from a dynamic linked list

*      Edit items in a dynamic linked list

*      Traverse a dynamic linked list

*      Design and implement basic test cases

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Compose C language programs

*      Compile a program using Microsoft Visual Studio 2019

*      Create basic test cases for a program

*      Apply arrays, strings, and pointers

*      Summarize differences between array notation and pointer notation

*      Apply pointer arithmetic

*      Apply basic string handling library functions

*      Define and implement structures in C

*      Summarize the operations of a linked list

**III. Overview & Requirements:**

In this assignment you will complete the Digital Music Manager that you started in [PA 2](https://eecs.wsu.edu/~aofallon/cpts122/progassignments/PA2.htm). You must implement the following features:

*      (4)  insert

*      (5)  delete

*      (7)  sort

*      (10) shuffle

You will also be required to write 3 test functions.

  *What must “insert” do?*

The “insert” command must prompt the user for the details of a new *record*. The prompt must request the artist name, album title, song title, genre, song length, number of times played, and rating. The new record must be *inserted* at the *front* of the list.

  *What must “delete” do?*

The “delete” command must prompt the user for a *song title*, and *remove* the matching record from the list. If the song title does *not* exist, then the list remains unchanged.

  *What must “sort” do?*

The “sort” command must prompt the user for 4 different methods to sort the *records* in the list. These include:

1.    Sort based on artist (A-Z)

2.    Sort based on album title (A-Z)

3.    Sort based on rating (1-5)

4.    Sort based on times played (largest-smallest)

Once a sort method is selected by the user, the sort must be performed on the records in the list. Consider using bubble sort, insertion sort, or selection sort.

  *What must “shuffle” do?*

The “shuffle” command must provide a random order in which the songs are played. This command must not modify the links in the list. It must just specify the order in which songs are played, based on the position of the song in the list. For example, let’s say we have a list with 5 songs at positions 1 – 5 in the list, shuffle must generate an order 1 – 5 in which the songs are played. An order 2, 5, 3, 1, 4 would require that the second song in the list is played first, the fifth song in the list is played second, the third song in the list is played third, the first song in the list is played fourth, and the fourth song in the list is played fifth. The songs are accessed by traversing the list both forwards and backwards to satisfy the order. Hence, the need for a doubly linked list!

Once again you will find an example musicPlayList.csv ([here](https://eecs.wsu.edu/~aofallon/cpts122/progassignments/musicPlayList.csv)).

  *What “test” functions are required?*

You must design and implement 3 test functions. These test functions must not accept any arguments or return any values. They should be self-sufficient. You should provide function declarations for them that are in a separate header file than your utility/application function declarations. Also, the corresponding implementations for them should be placed in a separate source file than your utility/applications function definitions and main (). You must implement one test function for *insert*, *delete*, and *shuffle* features for a total of 3 functions.

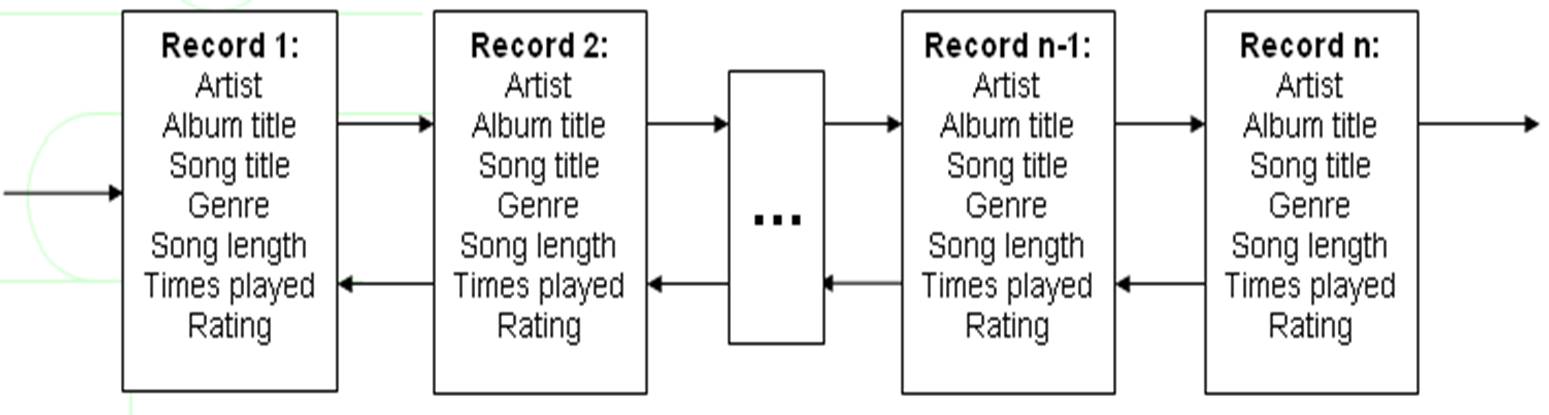
o   For the *insert* test function you must provide a test *case* with the following test *point*: artist name = “Perry, Katy”, album title = “Witness”, song title = “Chained to the Rhythm”, genre = “pop”, song length = “4:36”, times played = -1, rating = 6. List state = initially empty. You must think about what is your expected result? Should you able to insert a song with -1 times played? Should you able to add a song with rating 6? Is the head pointer of the list updated?

o   For the *delete* test function you must provide a test *case* with the following test *point*: song title to delete = “Chained to the Rhythm”. List state = artist name = “Perry, Katy”, album title = “Witness”, song title = “Chained to the Rhythm”, genre = “pop”, song length = “4:36”, times played = 3, rating = 5 (the only song in the list). You must think about what is your expected result? Has the head pointer been updated? Is it NULL? Did the memory get released?

o   For the *shuffle* test function you must provide a test *case* with the following test *point*: play order = 3, 1, 2. List state = you provide 3 songs. Does the shuffle play in the correct order?

**IV. Logical Block Diagram**

Once again, the logical block diagram for your doubly linked list should look like the following:



As you can see from the illustration a doubly linked list has a pointer to the next node and the previous node in the list. The first node’s previous node pointer is always NULL and the last node’s next pointer is always NULL. When you insert and delete nodes from a doubly linked list, you must always carefully link the previous and next pointers.

*BONUS:*

Modify your doubly linked list implementation(s) for your DMM so that last node in the list points to the first node, and the first node points to the last node. Hence, there is no longer a first or last node. This list is now called “circular”. Overall, it is called a circular doubly linked list. Any one of the nodes may by the current node!

**V. Submitting Assignments:**

1.       Using Blackboard Learn <https://learn.wsu.edu/webapps/login/> submit your assignment. You will submit your assignment in the ***lab*** Blackboard space. Under the "Content" link navigate to the "Programming Assignment Submissions" folder and upload your solution to the appropriate “Assignment” space. You must upload your solution, through an attachment, as <your last name>\_pa3.zip by the due date and time.

1. Your project must contain at least two header files (a .h file), three C source files (which must be .c files), and a local copy of the .csv file. One of the header files is required to contain the declarations for your test functions and one of the .c files must contain the implementations for those test functions.
2. Your project must build properly. The most points an assignment can receive if it does not build properly is 65 out of 100.

**VI. Grading Guidelines:**

This assignment is worth 100 points. Your assignment will be evaluated based on a successful compilation and adherence to the program requirements. We will grade according to the following criteria:

*      5 pts – Appropriate top-down design, style, and commenting according to class standards

*      17 pts – Correct “insert” command implementation

1.    (7 pts - 1pt/attribute) For prompting and getting the details of a *new* record from the user

2.    (10 pts) For correctly inserting the record at the *front* of the list

*      24 pts – For correct “delete” command implementation

1.    (3 pts) For prompting and getting the *song title* from the user

2.    (5 pts) For *searching* for specific record *matching* the song title

3.    (16 pts) For *removing* the matching record from the list, and reconnecting the list correctly

*       29 pts – Correct “sort” command implementation

1.    (3 pts) For prompting and getting the *sort method* from the user

2.    (7 pts) For sorting based on artist (A-Z)

3.    (7 pts) For sorting based on album title (A-Z)

4.    (6 pts) For sorting based on rating (1-5)

5.    (6 pts) For sorting based on times played (largest-smallest)

*       15 pts – Correct “shuffle” command implementation

1.    (5 pts) For generating the random order based on the number of songs in the list

2.    (10 pts) For moving through the list (forwards and backwards) and playing the songs in the order generated

*      10 pts – Robust *test* functions – 3 required

1.    (4 pts) For a test function that challenges the bounds of your *insert* feature.

2.    (3 pts) For a test function that challenges the bounds of your *delete* feature.

3.    (3 pts) For a test function that challenges the bounds of your *shuffle* feature.

*       BONUS: Up to 10 pts for correct circular implementation

**CptS 122 – Data Structures                                                                         **

**Lab 3: Data Structures and Dynamic Linked Stacks in C**

**Assigned:** Week of February 8, 2021

**Due:**At the end of the lab session

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design, implement and test a dynamic stack in C

*      Compare and contrast dynamic linked lists and dynamic stacks

*      Summarize the advantages of applying a stack within certain applications

*      Describe the operations applied to a stack including

1.    push ( )

2.    pop ( )

3.    top ( ) or peek ( )

4.    isEmpty ( )

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Compose a small C language program

*      Create test cases for a program

*      Apply and implement structs in C

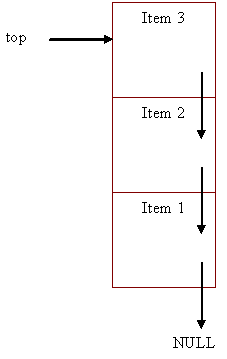
*      Apply and implement pointers in C

*      Apply and implement dynamic memory in C

*      Design and implement a dynamic singly linked list

**III. Overview & Requirements:**

This lab, along with your TA, will help you navigate through designing, implementing, and testing a dynamic stack. **Recall, a stack data structure is a restricted linked list, where only the top node in the stack may be accessed at any given time. A stack is referred to as a last-in, first-out (LIFO) structure as a result of this constraint. Furthermore, the operation of a stack must adhere to this restriction. A push ( ) operation adds a node to the top of the stack, a pop ( ) operation removes a node from the top of the stack, and a top ( ) or peek ( ) operation returns the data in the node at the top of the stack.** We will visualize a stack in the following way:



Labs are held in a “closed” environment such that you may ask your TA questions. Please use your TAs knowledge to your advantage. You are required to move at the pace set forth by your TA. Please help other students in need when you are finished with a task. You may work in pairs if you wish. However, I encourage you to compose your own solution to each problem. Have a great time! Labs are a vital part to your education in CptS 122 so work diligently.

**Tasks:**

**1.**For the following problem define a stackNode struct with data of type double. Implement the following operations for your stack data structure:

1.    isEmpty() – a predicate function which checks to see if the stack is empty; returns true if the stack is empty; false otherwise

2.    push() – inserts a node, with a double precision value, to the top of the stack; the node is allocated dynamically; the double precision value should be passed in as an argument, along with a double pointer to the top of the stack

3.    pop() – deletes a node from the top of the stack; accepts a double pointer to the top of the stack; does not return a value; this function should only be called after isEmpty ( ) is called

4.    top() or peek() – returns the data in the node at the top of the stack; does not modify the stack

**2.**Test your application. In the same project, create one more header file testStack.h and source file testStack.c (for a total of at least five files). The testStack.h file should contain function prototypes for test functions you will use on your stack functions. The testStack.c source file should contain the implementations for these test functions. You will have at least one test function per application function. For example, you will have an application function called push() (or a function very similar) that is used to insert a node on the top of the stack. In this task, you will need to create a test function called testPush() that passes in various double precision data directly into push() to see if it works correctly. Your test should check to see that the top node has the correct value. Does the top pointer change?

**3.**Tower of Hanoi: A very popular mathematical game or puzzle is referred to as the Tower of Hanoi. The idea behind the game is to find an efficient method for moving disks between *three* posts. Each disk has a different diameter, but all of them can be placed on the available posts. The goal of the game is to move all of the disks from one post to the another according the following rules:

1.    Only one disk may be transferred at a time

2.    Only the top disk on any post may be accessed at a given time

3.    No disk may be placed on top of a smaller disk at any point

At the start of the game, all of the disks must originally be placed such that the largest disk is on the bottom of the stack of one post, and the smallest is on the top of the stack on the same post. The disks should form a cone shape. Write a program to simulate the Tower of Hanoi game. For each move print the post number (1 – 3) from which the disk is taken, the diameter of the disk, and the resulting post on which the disk is placed. Also, show the current diameter of the disks on each post. You must use stacks to solve this problem! Initially start with three disks in your game. Note: if you visit <https://www.youtube.com/watch?v=3I3iVQUWLjo>, you will find an animation of how Tower of Hanoi should run.

**4.**Maze: Generate a maze with a start to end path. The maze be represented by a two dimensional array of integers, where a wall may be represented by a 0 and a door may be represented by a 1. Find a path in your maze by using a stack. Modify your stackNode to store Point data for a path in a maze. A Point should be defined as a struct with row and column fields. Whenever a fork in the maze is encountered, store the coordinates of the fork on the stack. If the current path does not provide a path to the end of the maze, then the last forking point can be popped and a different path may be taken. This is called backtracking.

**IV. Submitting Labs:**

*      You are not required to submit your lab solutions, unless you are unable to attend them synchronously. You should keep them in a folder that you may continue to access throughout the semester.

**V. Grading Guidelines:**

*      This lab is worth 10 points. Your lab grade is assigned based on completeness and effort. To receive full credit for the lab you must show up on time, continue to work on the problems until the TA has dismissed you, and complete at least 2/3 of the problems.

**CptS 122 – Data Structures                                                                                             **

**Programming Assignment 4: Basic Fitness Application in C++**

**Assigned:** Monday, February 22, 2021

**Due:**Friday, March 12, 2021 by midnight

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design, implement and test classes in C++

*      Declare and define *constructors*

*      Declare and define *destructors*

*      Compare and contrast *public* and *private* access specifiers in C++

*      Describe what is an *attribute* or data member of a class

*      Describe what is a *method* of a class

*      Apply and implement *overloaded* functions

*      Apply and implement overloaded *operators* (stream insertion and stream extraction)

*      Distinguish between pass-by-*value* and pass-by-*reference*

*      Discuss *classes* versus *objects*

*      Apply basic *file* operations on file *streams*

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Compose basic C++ language programs

*      Create basic test cases for a program

*      Apply arrays, strings, and pointers

**III. Overview & Requirements:**

You are to write a basic fitness application, in C++, that allows the user of the application to manually edit “diet” and “exercise” plans. For this application you will need to create three classes: *DietPlan*, *ExercisePlan,*and *FitnessAppWrapper*.

*Diet Plan Attributes:*

The class *DietPlan* is used to represent a *daily diet* plan. Your class must include three data members to represent your *goal* calories (an integer), plan *name* (a std::string), and *date* for which the plan is intended (a std::string). The maximum intake of calories for a day is stored in the *goal* calories.

*Exercise Plan Attributes:*

The class *ExercisePlan* is used to represent a *daily exercise* plan. Your class must include three data members to represent your *goal* steps (an integer), plan *name* (a std::string), and *date* for which the plan is intended (a std::string). Your *goal* steps represent the number of desired steps for a day.

*Diet and Exercise Plan Operations:*

Both the *DietPlan* and *ExercisePlan* should provide several *member* functions including: a constructor, copy constructor, and destructor. Remember that you will have to think about other appropriate member functions (think about *set*ter and *get*ter functions!). Member function *editGoal ()* should prompt the user for a new goal, and use the value to change the goal in the plan. Each time a plan is changed, the plan must be displayed to the screen, using an overloaded stream insertion operator (see below).

In the same file in which each class declaration exists, three *nonmember* functions must be declared. Note: an overloaded *operator* is considered an overloaded *function*. The overloaded stream insertion (<<) for both displaying a plan to the screen and for writing a plan to a file, and the extraction (>>) operator for reading a plan from a file.

**Observation: please notice that the *DietPlan*and *ExercisePlan*classes define very similar attributes and operations. In the future, we will be able to design around these similarities (using inheritance and polymorphism).**

*Fitness Application:*

Each of the daily plans will be read from a file. Each file must consist of exactly seven daily plans, representing a full week of plans. The daily diet plans will be read from a file called “dietPlans.txt” and the daily exercise plans will be read from a file called “exercisePlans.txt”. The format of the files must be represented in the following way:

*Plan name*

*Goal*

*Date in the form mm/dd/yyyy*

*(blank line)*

*Plan name*

*Goal*

*Date in the form mm/dd/yyyy*

You must read in each of the daily plans by applying an *overloaded* stream *extraction* operator: fileStream >> DietPlan or fileStream >> ExercisePlan. The overloaded operator must be defined as a *nonmember* function to the DietPlan and ExercisePlan classes. Each plan is stored into the next available position in your linear data structure whether it be an array, vector, or linked list.

**Observation: Inserting at the end of an array and vector requires (amortized) constant time. Inserting at the end of a linked list (with only a head pointer) requires linear time. Consider this idea as you develop your solution!**

The class *FitnessAppWrapper* is used to “wrap” the application. This class should contain two lists (must be an array, vector, or linked list) of weekly (7 days) plans: one diet and one exercise weekly plan. It must also contain two fstream objects (input/output file streams): one for each file. It must define the following member functions (some prototypes are given to you, but not all!):

-       void runApp (void): starts the main application.

-       void loadDailyPlan (fstream &fileStream, DietPlan &plan): must define two of these functions; one for a *DietPlan* and one for an *ExercisePlan*. This function reads one record from the given stream. These will be considered overloaded functions! Precondition: file is already open!

-       void loadWeeklyPlan (fstream &fileStream, DietPlan weeklyPlan[ ]): must define two of these functions; one for a *DietPlan* and one for an *ExercisePlan*. This function must read in all seven daily plans from the given stream. Note: the array parameter would change if using a vector or linked list! This function should call *loadDailyPlan ()* directly. Precondition: file is already open!

-       displayDailyPlan (): writes a daily plan to the screen. You must apply the overloaded stream insertion operator here! Note: you must determine the appropriate parameters and return type. Once again you must define two of these!

-       displayWeeklyPlan (): writes a weekly plan to the screen. This function must call *displayDailyPlan ()*. Note: you must determine the appropriate parameters and return type. Once again you must define two of these!

-       storeDailyPlan (): writes a daily plan to a file. You must apply the overloaded stream insertion operator here! Note: you must determine the appropriate parameters and return type. Once again you must define two of these!

-       storeWeeklyPlan (): writes a weekly plan to a file. This function must call *storeDailyPlan ()*. You must apply the overloaded stream insertion operator here! Note: you must determine the appropriate parameters and return type. Once again you must define two of these!

-       displayMenu (): displays nine menu options. These include:

1.    Load weekly diet plan from file.

2.    Load weekly exercise plan from file.

3.    Store weekly diet plan to file.

4.    Store weekly exercise plan to file.

5.    Display weekly diet plan to screen.

6.    Display weekly exercise plan to screen.

7.    Edit daily diet plan.

8.    Edit daily exercise plan.

9.    Exit.   // Note: this must write the most recent weekly plans to the corresponding files.

-       Other functions? There should be!

**Observation: Many of the functions in the FitnessAppWrapper class are overloaded. There’s one version for use on a DietPlan and one version for use on an ExercisePlan. We know these functions are considered overloaded because they have the same name, but different parameter types. In the future, we could use templates, and let the compiler generate code for us, instead of implementing several versions of the same function ourselves.**

BONUS:

Implement classes for *ListNode* and *List* to store the diet and exercise plans. You may need to implement a different linked list for each of the plans. In the future, this could be resolved by using templates.

**IV. Submitting Assignments:**

1.       Using Blackboard Learn <https://learn.wsu.edu/webapps/login/> submit your assignment. You will submit your assignment in the ***lab*** Blackboard space. Under the "Content" link navigate to the "Programming Assignment Submissions" folder and upload your solution to the appropriate “Assignment” space. You must upload your solution, through an attachment, as <your last name>\_pa4.zip by the due date and time.

1. Your project must contain at least three header files (.h files) and four C++ source files (which must be .cpp files). There should be one .h file per class declaration. There should be one .cpp for each set of operations belonging to a single class and one for the main () function.
2. Your project must build properly. The most points an assignment can receive if it does not build properly is 65 out of 100.

**V. Grading Guidelines:**

This assignment is worth 100 points. Your assignment will be evaluated based on a successful compilation and adherence to the program requirements. We will grade according to the following criteria:

*      5 pts – Appropriate top-down design, style, and commenting according to class standards

*      18 pts – Appropriate design and implementation of Class *DietPlan* (including member functions and data members)

1.    3 pts – 1 pt/each for declaring goal calories, plan name, and date

2.    2 pts – declaring and defining a constructor

3.    2 pts – declaring and defining a copy constructor

4.    1 pt – declaring and defining a destructor

5.    4 pts – declaring and defining setters/getters

6.    4 pts – declaring and defining an *editGoal* function

7.    2 pts – others?

*      18 pts – Appropriate design and implementation of Class *ExercisePlan* (including member functions and data members)

1.    3 pts – 1 pt/each for declaring goal steps, plan name, and date

2.    2 pts – declaring and defining a constructor

3.    2 pts – declaring and defining a copy constructor

4.    1 pt – declaring and defining a destructor

5.    4 pts – declaring and defining setters/getters

6.    4 pts – declaring and defining an *editGoal* function

7.    2 pts – others?

*      47 pts – Appropriate implementation of Class *FitnessAppWrapper*(including menu options, etc.)

1.    8 pts – 2 pts/each for declaring a list of diet plans, a list of exercise plans, a file stream associated with “dietPlans.txt”, and a file stream associated with “exercisePlans.txt”

2.    4 pts – declaring and defining *runApp* function

3.    4 pts – 2 pts/each for declaring and defining *loadDailyPlan* functions

4.    4 pts – 2 pts/each for declaring and defining *loadWeeklyPlan* functions

5.    4 pts – 2 pts/each for declaring and defining *displayDailyPlan* functions

6.    4 pts – 2 pts/each for declaring and defining *displayWeeklyPlan* functions

7.    4 pts – 2 pts/each for declaring and defining *storeDailyPlan* functions

8.    4 pts – 2 pts/each for declaring and defining *storeWeeklyPlan* functions

9.    2 pts – declaring and defining *displayMenu* function

10. 4 pts – opening and closing the files

11. 5 pts – others?

*      12 pts – 2 pts/each for the nonmember overloaded stream extraction and stream insertion operators (4 total stream insertion operators, 2 total stream extraction operators)

*      BONUS: Up to 10 pts – Linked list implementation using *ListNode* and *List*classes

**CptS 122 – Data Structures                                                                         **

**Lab 4: More Practice with Pointers, Lists, and Stacks in C**

**Assigned:** Week of February 15, 2021

**Due:**At the end of the lab session

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Compare and contrast dynamic linked lists and dynamic stacks

*      Distinguish between various types of pointers

*      Dive deeper into the <string.h> library and describe the purpose of memset(), memcpy(), memcmp(), and memchr()

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Compose a small C language program

*      Create test cases for a program

*      Apply and implement structs in C

*      Apply and implement pointers in C

*      Apply and implement dynamic memory in C

*      Design and implement a dynamic singly linked list

*      Design and implement a dynamic linked stack

**III. Overview & Requirements:**

This lab will allow you to further explore pointers and memory, linked lists, and stacks.

Labs are held in a “closed” environment such that you may ask your TA questions. Please use your TAs knowledge to your advantage. You are required to move at the pace set forth by your TA. You must work in teams assigned by your TA. However, I encourage you to compose your own solution to each problem. Have a great time! Labs are a vital part to your education in CptS 122 so work diligently.

**Tasks:**

1.    Given the following fragment of C code, answer the provided questions. If necessary, you could copy and paste the code into Visual Studio to answer the questions.

       Line 1: int n1 = 10, n2 = 42, list[] = {6, 8, 42, 3, 2, 2, -6};

       Line 2: int \* const p1 = &n1;

       Line 3: const int \* p2 = &n1;

Line 4: int \* p3 = list;

Line 5: const int \* const p4 = NULL;

Line 6: \*p1 = 15;

             Line 7: p1 = &n2;

             Line 8: p2 = &n2;

       Line 9: \*p2 = 67;

       Line 10: p3[4] = 67;

             Line 11: list = &n1;

Line 12: p4 = list;

       Line 13: \*p4 = 25;

a.    Is there any discernible difference between the declared types for p1 and p2 on lines 2 and 3?

b.    Is the assignment operation on line 6 legal?

c.    Is the assignment operation on line 7 legal?

d.    Is the assignment operation on line 8 legal?

e.    Is the assignment operation on line 9 legal?

f.     Is the assignment operation on line 10 legal?

g.    Is the assignment operation on line 11 legal?

h.    Is the assignment operation on line 12 legal?

i.     Is the assignment operation on line 13 legal?

2.    Complete the implementation for the following function. The function should allocate space on the heap for the given string parameter, and copy (use strncpy()) the string (pStr), including the null character to the heap. The function should return NULL if memory could not be allocated; otherwise it should return a pointer to the beginning of the allocated block of memory on the heap.

char \* copyStrToHeap(char \*pStr)

{

}

3.    Examine the following standard library functions, which are located in <string.h>. As you examine each one at the provided URL, please run the code examples in the C++ shell (click on the gear next to the example code).

a.    memset () - <http://www.cplusplus.com/reference/cstring/memset/>

b.    memcpy () - <http://www.cplusplus.com/reference/cstring/memcpy/>

c.    memcmp () - <http://www.cplusplus.com/reference/cstring/memcmp/>

d.    memchr () - <http://www.cplusplus.com/reference/cstring/memchr/>

In teams discuss the purpose of each function. Could you apply each one, independent of the provided code examples? Could you replace the strncpy() operation in Task 2 with a memcpy()? Could you use memcpy() to copy structs?

For the following problems define a struct node with data of type int.

4.    Implement a function that merges two linked lists in order. The function must be able to merge two ordered lists in descending or ascending order. The function accepts pointers to the two lists and an order to merge the lists, and returns a pointer to the beginning of the merged lists. Be sure to test your function! You may start with code that you have written for other labs. EXTRA CHALLENGE: Could you implement a merge lists function which merges n lists in order? This function would accept a pointer to an array of n pointers, where each pointer refers to a list.

5.    Implement a function that detects if there is a “loop” in a singly linked list. A “loop” is defined as a link from one node to itself or to another node that is before the node in the list. The function may only make one pass through the list. It must return 1 if a “loop” exists; 0 otherwise. Be sure to test your function! You may start with code that you have written for other labs. **This is a common interview question!**

6.    Implement a function that reverses a singly linked list in one pass of the list. The function accepts a pointer to the beginning of the list, and returns a pointer to the beginning of the reversed list. The function must not create any new nodes. Be sure to test your function! You may start with code that you have written for other labs. **This is a common interview question!**

7.    Implement a list using an array, instead of a linked implementation. With your team members discuss the following questions:

a.    Is it more efficient to delete the last node in an array or linked implementation of a list?

b.    Is it more efficient to delete the first node in an array or linked implementation of a list?

c.    Is it more efficient to delete a node, in general, in an array or linked implementation of a list?

d.    Is it more efficient to insert a node at the end in an array or linked implementation of a list?

e.    Is it more efficient to insert a node at the front in an array or linked implementation of a list?

f.     Is it more efficient to insert a node, in general, in an array or linked implementation of a list?

g.    Is it more efficient to access a node, in general, in an array or linked implementation of a list?

8.    This problem is from lab 3. If you did not complete it, then use your stack code from lab 3 to solve the following problem. Maze: Generate a maze with a start to end path. The maze be represented by a two dimensional array of integers, where a wall may be represented by a 0 and a door may be represented by a 1. Find a path in your maze by using a stack. Modify your stackNode to store Point data for a path in a maze. A Point should be defined as a struct with row and column fields. Whenever a fork in the maze is encountered, store the coordinates of the fork on the stack. If the current path does not provide a path to the end of the maze, then the last forking point can be popped and a different path may be taken. This is called backtracking.

**IV. Submitting Labs:**

****You are not required to submit your lab solutions, unless you are unable to attend them synchronously. You should keep them in a folder that you may continue to access throughout the semester.

**V. Grading Guidelines:**

       This lab is worth 10 points. Your lab grade is assigned based on completeness and effort. To receive full credit for the lab you must show up on time, continue to work on the problems until the TA has dismissed you, and complete at least 2/3 of the problems.

CptS**122 – Data Structures                                                                                             **

**Programming Assignment 5: Grocery Store Simulation**

**Assigned:** Friday, March 12, 2021

**Due:**Monday, March 22, 2021 by midnight (extended date)

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design and implement a dynamic queue

*      Allocate and de-allocate memory at runtime

*      Manipulate links in a dynamic queue

*      Insert items into a dynamic queue

*      Delete items from a dynamic queue

*      Traverse a dynamic queue

*      Design, implement, and apply test cases in C++

*      Design test classes in C++

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Compose C++ language programs

*      Create basic test cases for a program

*      Apply arrays, strings, and pointers

*      Summarize differences between array notation and pointer notation

*      Apply pointer arithmetic

*      Apply basic string handling library functions

*      Define and implement classes in C++

*      Summarize the operations of a linked list

*      Describe the operations of a queue including: enqueue ( ), dequeue ( ), printQueue ( )

**III. Overview & Requirements:**

Note: parts of this assignment were inspired by Deitel and Deitel’s Supermarket Simulation problem. We’ve all had the pleasant experience of standing in line at the grocery store. As I’m standing in line I’m always trying to figure out if I chose the line with the fastest service. In most cases, I fail miserably. Let’s write a program to simulate two lines in a grocery store, which will allow us to better understand how to select the fastest line. Note: you’ll be required to write some test cases for this program. I **highly** recommend that you start with your tests before you implement any other aspect of the program. Starting with your tests will also allow for you to better design and implement your main application code.

For this assignment you will need to simulate two lines using queues. This will require that you develop enqueue ( ) (insert), dequeue ( ) (delete), and printQueue ( ) operations for a queue. Although you will instantiate two queues, each one of these will consist of the same kind of queue nodes. Define a queue node in the following manner:

class Data

{

public: // Member functions

private:

     int customerNumber; // Unique identifier; starts at 1; after 24 hours should be reset to 1

     int serviceTime;   // Random time; varies between express and normal lanes; units in minutes

     int totalTime;     // totalTime = serviceTime + sum of serviceTimes of customers in line before this customer; units in minutes

}; // This memory needs to be allocated on the heap!

class QueueNode

{

public: // Member functions

private:

     Data \*pData;    // The memory for Data will need to be allocated on the heap as well!

     QueueNode \*pNext;

};

You must also define a queue in the following manner:

class Queue

{

public: // Member functions

private:

     QueueNode \*pHead;

     QueueNode \*pTail;

};

One of your queues will represent the express lane and the other a normal lane. You will randomly generate arrival times and service times of customers into each lane. The express lane has customers arrive every one to five minutes, and customers arrive every three to eight minutes in the normal lane. Service times vary from one to five minutes, and three to eight minutes, for express and normal lane customers, respectively. As customers arrive into each line print out a message indicating in which line each customer arrives, along with the overall corresponding arrival time and customer number. When customers have finished checking out, print out a message indicating which line the customer was in, along the corresponding customer number and totalTime in the line. Allow for the simulation to run for n number of minutes, where n is inputted by the user.

The general program flow is as follows:

1. Generate a random number between 1 – 5 and 3 – 8 for express and normal lanes, respectively. This represents the arrival time of the first customer into each lane. Set the variable for total time elapsed to 0.
2. As customers arrive into each line, randomly generate a service time for each. Start processing the customers in the lanes based on service time. Randomly generate the arrival time of the next customer into each line. Elapsed time should be updated by one unit.
3. As each minute elapses, a new customer may arrive and/or another customer may be done checking out. Display the appropriate messages as described above.
4. For every 10 minutes, print out the entire queue for each line
5. Repeat steps 2 through 4 for n minutes of simulation.

Hints: since this is a simulation one minute is really one unit of time. Thus, the incrementing of an integer variable could represent one minute of time elapsing.

*Required test cases:*

Declare and define a test *class* for your application. You must declare your test class in a separate .h file from your other classes. You must also place your test case implementations in a separate .cpp file from your other classes. You must implement a total of 5 test cases. You must write the following test cases:

- One test case that executes your enqueue() operation on an empty queue

- One test case that executes your enqueue() operation with one node in the queue

- One test case that executes your dequeue() operation on a queue with one node

- One test case that executes your dequeue() operation on a queue with two nodes

- One test case that runs your simulation for 24 hours

BONUS:

Modify QueueNode such that it contains a pointer to the start of a dynamic singly linked list. The linked list will consist of grocery items corresponding to one person. These items should be strings like “cereal”, “milk”, “steak”, etc. Adjust the serviceTime of the QueueNode so that it is no longer random, but proportional to the number of items for the person served.

**IV. Submitting Assignments:**

1.       Using Blackboard Learn <https://learn.wsu.edu/webapps/login/> submit your assignment. You will submit your assignment in the ***lab*** Blackboard space. Under the "Content" link navigate to the "Programming Assignment Submissions" folder and upload your solution to the appropriate “Assignment” space. You must upload your solution, through an attachment, as <your last name>\_pa5.zip by the due date and time.

1. Your project must contain at least two header files (a .h file) and three C++ source files (which must be .cpp files).
2. Your project must build properly. The most points an assignment can receive if it does not build properly is 65 out of 100.

**V. Grading Guidelines:**

This assignment is worth 100 points. Your assignment will be evaluated based on a successful compilation and adherence to the program requirements. We will grade according to the following criteria:

*      5 pts – Appropriate top-down design, style, and commenting according to class standards

*      15 pts – Correct usages of two queues in simulation

*      5 pts – Correct Data class

*      5 pts – Correct QueueNode class

*      7 pts – Correct Queue class

*      10 pts – Correct constructors/destructors for the classes

*      5 pts – Correct getters/setters for the classes

*      5 pts – Correct printQueue()

*      3 pts – Correct isEmpty()

*      15 pts – Correct enqueue()

*      15 pts – Correct dequeue()

*      10 pts (2 pts/test case) – Correct test cases for the application

*      BONUS: Up to 20 pts for implementation of grocery item list/customer

**CptS 122 – Data Structures                                                                         **

**Lab 5: The Wonderful World of C++, Classes, and Objects**

**Assigned:** Week of February 22, 2021

**Due:**At the end of the lab session

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design, implement and test classes in C++

*      Declare and define *constructors*

*      Declare and define *destructors*

*      Compare and contrast *public* and *private* access specifiers in C++

*      Describe what is an *attribute* or data member of a class

*      Describe what is a *method* of a class

*      Apply and implement *overloaded* functions

*      Distinguish between pass-by-*value* and pass-by-*reference*

*      Discuss *classes* versus *objects*

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Compose a small C++ language program

*      Create test cases for a program

**III. Overview & Requirements:**

This lab will allow you to further explore lists, and navigate through designing, implementing, and testing classes in C++.

Labs are held in a “closed” environment such that you may ask your TA questions. Please use your TA’s knowledge to your advantage. You are required to move at the pace set forth by your TA. Please help other students in need when you are finished with a task. Have a great time! Labs are a vital part to your education in CptS 122 so work diligently.

**Tasks:**

1.    In teams, before you implement any code, discuss, with the use of the whiteboard, a general C++ class that will satisfy the requirements for the application described below. Complete programming project Complex numbers from your Deitel and Deitel C++ How to Program book, with some additional tasks provided by me.

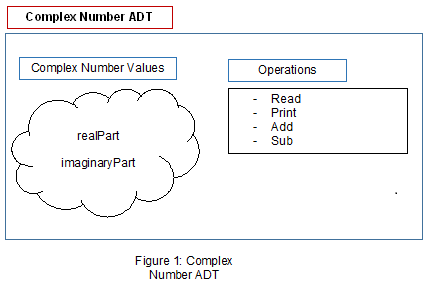
     Create a class called *Complex* for performing arithmetic with complex numbers. Write a program to test your class. Complex numbers have the form:

            realPart + imaginaryPart \* i

     where *i* is

                sqrt (-1)

     use double variables to represent the private data of the class. Provide a constructor that enables an object of this class to be initialized when it is declared. The constructor should contain default values in case no initializers are provided. Define *public* setters and getters to access the private data members. Below in Figure 1 is a summary of the Complex Number ADT:



Also, provide functions that perform the following tasks:

a)    Adding two *Complex* numbers: The real parts are added together and the imaginary parts are added together. Implement the addition operation in three ways:

i.     Implement a *member* function of class *Complex* called add() that must do the following: accept one *Complex* number *rhs* for an argument, add *rhs* to the data members in the object that invokes the add() function, and return the result.

ii.    Implement a *non-member* function called add(). Make sure it has the same name as the one defined in part (i). Place the prototype/declaration for this function outside of the *Complex* number class’s declaration (below the class declaration). Place its definition inside of the Complex.cpp file. The function must do the following: accept two *Complex* numbers called *lhs* and *rhs* for arguments, add *lhs* and *rhs* together, and return the result.

iii.  Implement a *non-member* overloaded addition (+) *operator*. Place the prototype/declaration for this function outside of the *Complex* number class’s declaration (below the class declaration). Place its definition inside of the Complex.cpp file. The function must do the following: accept two *Complex* numbers called *lhs* and *rhs* for arguments, add *lhs* and *rhs* together, and return the result. Note: the overloaded + is a binary operation, which requires two arguments! Use the following prototype/declaration:

Complex operator+ (const Complex &lhs, const Complex &rhs);

     Note: overloading the + operator allows for us to use statements such as: c3 = c1 + c2, where c1, c2, and c3 are *Complex* numbers. Another interpretation is: c3 = operator+(c1, c2).

Now place the following statements in a test driver, and use the debugger to watch each of the *Complex* numbers:

          Complex c1(3.5, 2), c2(5.5, 7), c3;

          c3 = c1.add(c2);  // member add () function

          c3 = add(c1, c2); // non-member add () function

          c3 = c1 + c2;       // overloaded + operator

b)    Subtracting two *Complex* numbers: The real part of the right operand is subtracted from the real part of the left operand, and the imaginary part of the operand. Solve this problem by overloading the subtraction (–) *operator*. Use the debugger to watch each of the *Complex* numbers declared for part (a).

          c3 = c1 - c2;       // overloaded - operator

c)    Reading *Complex* numbers from the keyboard, in the form a + bi, where *a* is the real part and *b* is the imaginary part. Implement the read operation in two ways:

i.     Implement a *member* function of class *Complex* called read() that must do the following: accept no arguments, read in the real and imaginary parts of the number from the standard *input* stream in the form:

 a + bi,

and return nothing. Yes, you must read in the + (or minus -) and the i, but they should be discarded. This function does not prompt for the *Complex* number. The prompt is done external to the function.

ii.    Implement a *non-member* overloaded stream extraction (>>) *operator*. Place the prototype/declaration for this function outside of the *Complex* number class’s declaration (below the class declaration). Place its definition inside of the Complex.cpp file. The function must do the following: accept one *istream*object called *lhs* and one *Complex* number called *rhs* for arguments, extract the real and imaginary parts of the number from the standard *input* stream in the form:

 a + bi,

and return the i*stream* object (so we can chain >> together!). Note: once again, you should discard the the + (or minus -) and the i. Use the following prototype/declaration:

istream & operator>> (istream &lhs, Complex &rhs);

Assuming that you still have the same Complex numbers instantiated: c1, c2, and c3. Use the debugger to watch each of the *Complex* numbers declared. Also, place the following statements in a test driver:

cout << “Enter a complex number in the form a + bi: “;

c1.read();

cout << “Enter a complex number in the form a + bi: “;

c2.read();

cout << “Enter two complex numbers in the form a + bi (each separated by whitespace): “;

cin >> c1 >> c2;

d)    Printing *Complex* numbers to the screen, in the form a + bi, where *a* is the real part and *b* is the imaginary part. Implement the print operation in two ways:

j.     Implement a *member* function of class *Complex* called print() that must do the following: accept no arguments, insert the real and imaginary parts of the number into the standard *output* stream in the form:

 a + bi,

and return nothing.

jj.   Implement a *non-member* overloaded stream insertion (<<) *operator*. Place the prototype/declaration for this function outside of the *Complex* number class’s declaration (below the class declaration). Place its definition inside of the Complex.cpp file. The function must do the following: accept one *ostream*object called *lhs* and one *Complex* number called *rhs* for arguments, insert the real and imaginary parts of the number into the standard *output* stream in the form:

 a + bi,

and return the *ostream* object (so we can chain << together!). Use the following prototype/declaration:

ostream & operator<< (ostream &lhs, const Complex &rhs);

          Once again, assuming that you still have the same Complex numbers instantiated: c1, c2, and c3. Place the following statements in a test driver:

                              c1.print();

                              c2.print();

                              c3.print();

                              cout << c1 << “ “ << c2 << “ “ << c3 << endl;

2.    Read the following <http://www.consumerfinance.gov/askcfpb/309/what-is-a-credit-report.html>. Once again work with your team to write and test a class called CreditReport. This class represents a real-world credit report. In this problem we will model the credit report in the following way. It should contain a credit score (ranges 330 – 830), a debt profile (total real estate and credit card debt), account types (number of real estate, credit cards, and retail cards), length of history (oldest account age and average account age), and number of hard inquiries (number of times your credit report has been accessed) as attributes. Operations that may be applied to your CreditReport include: printReport and updateReport (you may refine updateReport for use with individual class attributes). Be sure to include all necessary getters and setters, constructors, and destructors. Be sure to define a copy constructor! Write an application which instantiates three credit reports. These include Experian, TransUnion, and Equifax. Your application should decrease credit scores as the credit limit on cards is approached and increase scores as the credit is paid off. Also the older the credit accounts, the higher the credit score. Also, use the other attributes of the CreditReport as you see fit.

**IV. Submitting Labs:**

****You are not required to submit your lab solutions, unless you are unable to attend them synchronously. You should keep them in a folder that you may continue to access throughout the semester.

**V. Grading Guidelines:**

       This lab is worth 10 points. Your lab grade is assigned based on completeness and effort. To receive full credit for the lab you must show up on time, continue to work on the problems until the TA has dismissed you, and complete at least 2/3 of the problems.

**CptS 122 – Data Structures                                                                                             **

**Programming Assignment 6: Morse Code Lookup BST**

**Assigned:** Friday, March 19, 2021

**Due:**Friday, March 26, 2021 by midnight

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design, implement, and test a Binary Search Tree (BST)

*      Apply a BST for looking up Morse Codes

*      Discuss *classes* versus *objects*

*      Implement *container* classes

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Compose basic C++ language programs

*      Create basic test cases for a program

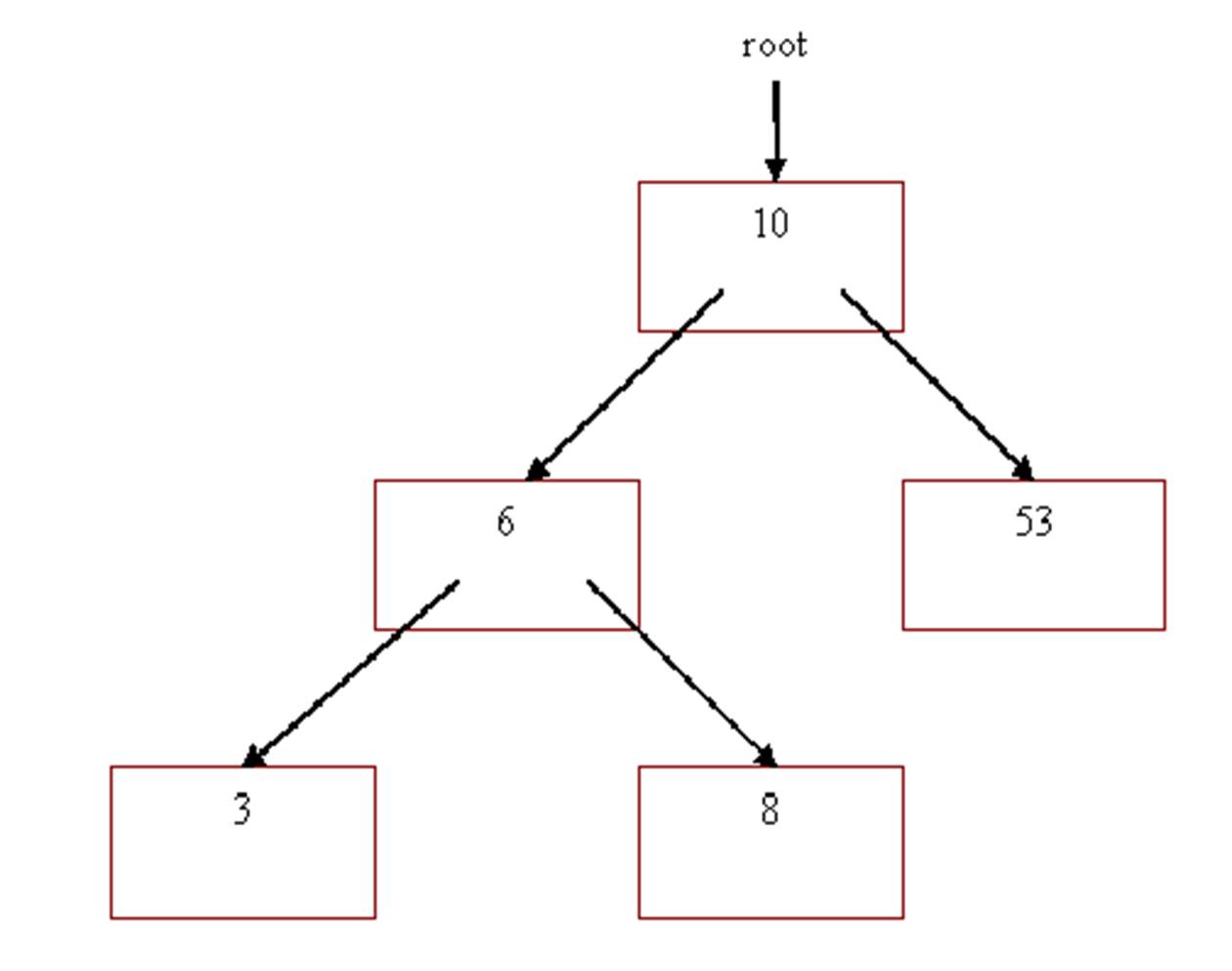
*      Apply arrays, strings, and pointers

*      Design, implement, and apply classes

*      Design, implement, and apply linked lists

**III. Overview & Requirements:**

Recall, a Binary Search Tree (BST) data structure is a nonlinear data structure. A BST is traversed by starting at the root pointer. The root node is the first node inserted into the tree. Nodes are inserted into the tree such that all items to the left of the root node are less than, and all items to the right of the root are greater than its item. Also, this property holds true for any particular node in the tree. We will visualize a BST in the following way:



In this assignment you will be using a BST to convert English characters to Morse code strings. Morse code is a famous coding scheme that assigns a series of dots and dashes to each letter of the alphabet, each digit, and a few special characters. In sound-oriented systems, the dot represents a short sound and the dash represents a long sound. Other representations of dots and dashes are used with light-oriented systems and signal-flag systems (from Deitel and Deitel C How to Program).

**1. (15 pts) Defining the BSTNode structure**

For the first part of the assignment, you should start by designing the BSTNode class for the BST. Create a class for the BSTNode data that will have as its members a character and a string. The character will hold the English text character, and the string will hold the corresponding Morse code characters for that English text character. You should also define left and right child pointers that point to BSTNode objects. You must have a constructor that accepts arguments to set the English text character and Morse code string.

**2. (50 pts) Create the BST code and create a Morse *lookup* BST**

Next, you should be able to read in the Morse table from a file called “MorseTable.txt”. You should rearrange the Morse table in the file to make sure that your lookup tree is balanced. I recommend that you diagram a tree that provides a balanced tree so that you know how to order your “MorseTable.txt” file. Think about the order of insertions. However, the tree does not have to balance itself.

The tree should be built by the constructor for the BST. This means the constructor must open and read the file, create nodes for each character in the tree, insert the nodes into the tree (using an insert () function), and close the file. Note: the tree object could be declared as const, since all changes to it are being performed in the constructor. However, if you declare your object as a const, be sure to also declare your print () and search () functions as const. You should arrange the tree so that it is alphabetically ordered from left to right. Create a print ( ) function that uses recursion to traverse the tree in order (left most printed first). Also, build a search ( ) function that will return the Morse code string for each English character searched for. Do you need to return a found indicator from the search function? Should you use recursion? Finally, implement a destructor, which destroys the entire tree.

**Morse Code Alphabet**:

|  |  |  |
| --- | --- | --- |
| **A   .-**  **B   -...**  **C   -.-.**  **D   -..**  **E   .**  **F   ..-.**  **G   --.**  **H   ....**  **I   ..**  **J   .---**  **K   -.-**  **L   .-..**  **M   --** | **N   -.**  **O   ---**  **P   .--.**  **Q   --.-**  **R   .-.**  **S   ...**  **T   -**  **U   ..-**  **V   ...-**  **W   .--**  **X   -..-**  **Y   -.--**  **Z   --..** | **0   -----**  **1   .----**  **2   ..---**  **3   ...--**  **4   ....-**  **5   .....**  **6   -....**  **7   --...**  **8   ---..**  **9   ----.**  **FULLSTOP    .-.-.-**  **Comma ‘,’   --..--**  **Query ‘?’   ..--..** |

**3. (30 pts) Putting the pieces together**

First, print the current tree. Next, you must open a file called “Convert.txt”, which consists of English alphabetic characters, spaces, commas, and periods. You must “look” for each English character with a search ( ) function on the BST, and print the Morse code string for that character. For each character in “Convert.txt”, convert the character to a Morse code string. Each Morse character in the string will be separated by a space. Each complete Morse string will be separated by three spaces. Each newline character will be echoed to the screen. Note: you should convert any lowercase English characters to uppercase before processing the English text.

Below is an example test file (you should add more characters to test all conversions!):

(Convert.txt)

This is a test of the cpts 122

Morse code conversion tool.

(Echoed to screen)

**- .... .. ...   .. ...   .-   - . ... -   --- ..-.   - .... .   -.-. .--. - ...   .---- ..--- ..---**

**-- --- .-. ... .   -.-. --- -.. .   -.-. --- -. ...- . .-. ... .. --- -.   - --- --- .-.. .-.-.-**

**BONUS (15 pts):** Implement a BSTNode and BST class template. Think about: how do you accommodate two different types in a class template?

**IV. Submitting Assignments:**

1.       Using Blackboard Learn <https://learn.wsu.edu/webapps/login/> submit your assignment. You will submit your assignment in the ***lab*** Blackboard space. Under the "Content" link navigate to the "Programming Assignment Submissions" folder and upload your solution to the appropriate “Assignment” space. You must upload your solution, through an attachment, as <your last name>\_pa6.zip by the due date and time.

1. Your project must contain at least two header files (.h files) and three C++ source files (which must be .cpp files).
2. Your project must build properly. The most points an assignment can receive if it does not build properly is 65 out of 100.

**V. Grading Guidelines:**

This assignment is worth 100 points. Your assignment will be evaluated based on a successful compilation and adherence to the program requirements. We will grade according to the following criteria:

*      5 pts – Appropriate top-down design, style, file organization (each class should be in its own .h file), and commenting according to class standards

*      15 pts - Defining the BSTNode structure

1.    4 pts correct data members in node (char, string, left and right pointers)

2.    3 pts correct constructor

3.    8 pts other member functions

*      50 pts - Creating the BST code and create a Morse *lookup* BST

1.    2 pts correct data member in BST (root)

2.    10 pts correct insert ()

3.    2 pts for opening “MorseTable.txt”

4.    2 pts for closing “MorseTable.txt”

5.    6 pts for reading contents of “MorseTable.txt”

6.    5 pts for correct print ()

7.    9 pts for correct search ()

8.    6 pts for correct constructor

9.    8 pts for correct destructor

*      30 pts -  Putting the pieces together

1.    5 pts for printing the tree

2.    2 pts for opening “Convert.txt”

3.    2 pts for closing “Convert.txt”

4.    6 pts for reading contents of “Convert.txt”

5.    10 pts for performing conversion of English to Morse code

6.    5 pts for echoing Morse code to screen

*      **BONUS 15 pts** – Working BSTNode and BST class template

**CptS 122 – Data Structures                                                                         **

**Lab 6: Data Structures, Dynamic Linked Queues, and Unit Tests in C++**

**Assigned:** Week of March 1, 2021

**Due:**At the end of the lab session

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design, implement and test a dynamic queue implemented with a singly linked list in C++

*      Compare and contrast dynamic linked lists, dynamic linked stacks, and dynamic linked queues

*      Summarize the advantages of applying a queue within certain applications

*      Describe the operations applied to a queue including

1.    enqueue ( )

2.    dequeue ( )

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Compose a small C++ language program

*      Create test cases for a program

*      Apply and implement classes in C++

*      Apply and implement pointers in C/C++

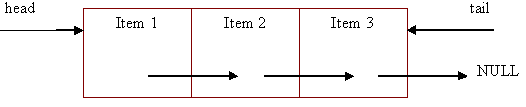
*      Apply and implement dynamic memory in C++

*      Design and implement a dynamic singly linked list

*      Design and implement a dynamic stack implemented with a singly linked list

**III. Overview & Requirements:**

This lab, along with your TA, will help you navigate through designing, implementing, and testing a dynamic queue implemented with a linked list. **Recall, a queue data structure is a restricted linked list, where only the front or head node in the queue may be processed and then removed, at any given time. However, only nodes may be added to the end, back, or tail of the queue. A queue is referred to as a first-in, first-out (FIFO) structure as a result of this constraint.** Furthermore, the operations of a queue must adhere to this restriction. An enqueue() operation adds a node to the tail of the queue and a dequeue() operation removes a node from the head of the queue. We will visualize a queue in the following way:



Labs are held in a “closed” environment such that you may ask your TA questions. Please use your TAs knowledge to your advantage. You are required to move at the pace set forth by your TA. Please help other students in need when you are finished with a task. You may work in pairs if you wish. However, I encourage you to compose your own solution to each problem. Have a great time! Labs are a vital part to your education in CptS 122 so work diligently.

**Tasks:**

**1.**For the following problem define a class QueueNode with data of type std::string. Also define a class Queue with two pointers: one for the *head* and one for the *tail*. Both pointers are QueueNode \* types. Implement the following operations for your queue data structure:

1.    isEmpty() – a predicate function which checks to see if the queue is empty; returns true if the queue is empty; false otherwise

2.    enqueue() – inserts a node into the queue at the tail; the node is allocated dynamically on the heap using the C++ operator new; returns true if the memory was allocated for a node, false otherwise

3.    dequeue() – deletes a node from the head of the queue using the C++ operator delete; returns the data in the node; precondition: queue is not empty (isEmpty () must be called before dequeue () is called)

4.    printQueueRecursive() – recursively prints out the data in the queue

You should also consider any constructors/destructors, overloaded operators, and getters/setters that you need.

**2.**Test your application. In the same project, create one more header file testQueue.h and source file testQueue.cpp (for a total of at least five files). The testQueue.h file should contain function prototypes for test functions you will use on your queue functions. The testQueue.cpp source file should contain the implementations for these test functions. You will have at least one test function per application function. For example, you will have an application function called enqueue() (or a function very similar) that is used to insert a node into the queue at the tail. In this task, you will need to create a test function called testEnqueue() that passes in various data directly into enqueue() to see if it works as intended. You will also want to test these functions on empty and non-empty queues.

**3.**Work on the current programming assignment.

**IV. Submitting Labs:**

*      You are not required to submit your lab solutions, unless you are unable to attend them synchronously. You should keep them in a folder that you may continue to access throughout the semester.

**V. Grading Guidelines:**

*      This lab is worth 10 points. Your lab grade is assigned based on completeness and effort. To receive full credit for the lab you must show up on time, continue to work on the problems until the TA has dismissed you, and complete at least 2/3 of the problems.

CptS**122 – Data Structures                                                                                             **

**Programming Assignment 7: Attendance Tracker w/ class Templates**

**Assigned:** Monday, March 29, 2021

**Due:**Friday, April 9, 2021 by midnight

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design, implement and test classes in C++

*      Apply class templates in C++

*      Declare and define *constructors*

*      Declare and define *destructors*

*      Compare and contrast *public* and *private* access specifiers in C++

*      Describe what is an *attribute* or data member of a class

*      Describe what is a *method* of a class

*      Apply and implement *overloaded* functions

*      Distinguish between pass-by-*value* and pass-by-*reference*

*      Discuss *classes* versus *objects*

*      Implement *container* classes

*      Implement a *list* class

*      Implement a *stack* class

*      Read and write files in C++

*      Programmatically acquire calendar dates

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Compose basic C++ language programs

*      Create basic test cases for a program

*      Apply arrays, strings, and pointers

**III. Overview & Requirements:**

Let us create an application that manages attendance for a course. This application has four major requirements:

Requirement 1 (Import records): The application must import records pertaining to each student registered for the course from a course list.

Requirement 2 (Mark absences): The application must allow the user to mark each student in the course as present or absent on any given day.

Requirement 3 (Generate reports): The application must generate reports based on criteria.

Requirement 4 (Menu): The application must support a user interface to the attendance tracker.

*Import records:* Records must be read from a comma-separated values (.csv) course file. A .csv file stores data as plaintext in tabular form. Each row in the file is considered a *record*. Each record consists of *fields* separated by commas. Please start with this [.csv file](https://eecs.wsu.edu/~aofallon/cpts122/progassignments/classList.csv). In this assignment the following fields will be present for each record:

-       record number (max 3 digits)

-       ID number (max 9 digits)

-       name (last, first)

-       email

-       units (number of credits for class or AU for audit)

-       program (major)

-       level (freshman, sophomore, junior, senior, graduate)

You are required to use a dynamic singly linked list to store student records. As each record is imported from the file, the record must be inserted at the front of the list. Inserting at the front of a dynamic linked list is very efficient (constant time – O(1)). You are required to implement two class *templates* for the list, plus an additional two classes (not required to be class templates) for the Data and Stack. Each of the class templates will only require that *one* type is used: type T. One class template is the *Node*class, which contains a data member of type T (this will be replaced by the type class Data) when it is instantiated, along with a pointer to the next Node. The *Data* class stores the fields acquired from each record. In addition to the fields in the file, you are required to add two extra fields in to the class Data. These fields include *number* of absences and a *stack*(must be implemented using an array or std::vector) for storing the dates of absences. Remember, class Data is not a template!!! The most recent absence date will always be at the top (Last-In First-Out, LIFO)! The second class template is the *List* class, which is a *container* for the Nodes. The List class will be considered your *master* list. Lastly, you are required to implement only one class for the Stack. The *Stack* class will be implemented using an array or std::vector. The Stack class must support push (), pop (), peek (), and isEmpty () operations, but does not have to be a template. All of the stack operations should execute in constant time (O(1)).

*Mark absences:* The user of the program should be able to view the *master* list of students in the course and mark absences for the current day. This may be implemented by simply traversing the linked list (linear time (O(n)) and asking is the student absent? Yes or no? The date for the day must be derived from the computer’s date. The following fragment of code illustrates how to derive the date from the computer:

       // retrieved from stackoverflow - http://stackoverflow.com/questions/997946/how-to-get-current-time-and-date-in-c

       time\_t t = time(0);   // get time now

       struct tm \* now = localtime( & t );

       cout << (now->tm\_year + 1900) << '-'

            << (now->tm\_mon + 1) << '-'

            << now->tm\_mday

            << endl;

*Generate reports:* The user of the program should be able to generate two versions of reports. One version is a report that shows all of the students in the class and the number of times they have been absent, along with the date of the most recent absence (peek ()). A second version is a report that provides only the names of the students absent for those who are absent greater than some threshold set by the user. You do NOT need to show the dates absent for the second version. Write each report to a different .txt file. What are the time complexities or Big-O of the generate report algorithms?

*Menu:*At startup of the program a menu must be displayed. The menu must support six options. These include:

1.    Import course list

2.    Load master list

3.    Store master list

4.    Mark absences

5.    BONUS: Edit absences

6.    Generate report

7.    Exit

Option 1: Reads the .csv course file and overwrites the master list.

Option 2: Populates the master list with previous nodes from master.txt.

Option 3: Stores the contents of the master list’s nodes to master.txt.

Option 4: Runs through the master list, displays each student’s name, and prompts if he/she was absent for the current day. The data must be pushed to the stack that is contained within the node representative of the student.

BONUS: Option 5: Prompts for an ID number or name of student to edit. Prompts for the date of absence to edit.

Option 6: Leads to submenu -> 1. Generate report for all students; showing only the most recent absence for each student. This is a peek () operation! 2. Generate report for students with absences that match or exceed (the number entered by the user).

Option 7: Exit the program.

You are required to define a class for your menu, which is NOT a template.

**BONUS:** Edit absences - The user of the program should be able to access each student’s record and edit absences. A search (linear time) through the master list based on student ID or name must be supported. If a student was initially marked absent for a date, but later was determined to be present, then the absence should be removed from the record. This includes updating the number of absences field. Be sure to add an Edit option to your menu!

**IV. Submitting Assignments:**

1.    Using Blackboard Learn <https://learn.wsu.edu/webapps/login/> submit your assignment. You will submit your assignment in the ***lab*** Blackboard space. Under the "Content" link navigate to the "Programming Assignment Submissions" folder and upload your solution to the appropriate “Assignment” space. You must upload your solution, through an attachment, as <your last name>\_pa7.zip by the due date and time.

1. Your project must build properly. The most points an assignment can receive if it does not build properly is 65 out of 100.

**V. Grading Guidelines:**

This assignment is worth 100 points. Your assignment will be evaluated based on a successful compilation and adherence to the program requirements. We will grade according to the following criteria:

*      5 pts – Appropriate top-down design, style, and commenting according to class standards

*      30 pts – Appropriate design and implementation of class *templates* *Node*and*List* (including member functions and data members)

*      12 pts – Appropriate design and implementation of class *Stack* (including member functions and data members)

*      8 pts – Appropriate design and implementation of class *Data*(including member functions and data members)

*      15 pts – Working “Import records” feature

*      10 pts – Working “Mark absences” feature

*      10 pts – Working “Generate reports” feature – each report is 5 pts

*      10 pts - Working “Menu” feature encapsulated by a menu object – this is not a class template

*      **BONUS 20 pts** – Working “Edit absences” feature

**CptS 122 – Data Structures                                                                         **

**Lab 7: Developing a (Linked) List Class in C++**

**Assigned:** Week of March 8, 2021

**Due:**At the end of the lab session

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design, implement, and test a List class in C++

*      Apply a customized linked List class

*      Compare and contrast shallow versus deep copy

*      Compare and contrast value classes versus container classes

*      Apply and implement *overloaded* functions and operators

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Create test cases for a program

*      Design, implement and test classes in C++

*      Declare and define *constructors*

*      Declare and define *destructors*

*      Compare and contrast *public* and *private* access specifiers in C++

*      Describe what is an *attribute* or data member of a class

*      Describe what is a *method* of a class

*      Apply and implement *overloaded* functions

*      Distinguish between pass-by-*value* and pass-by-*reference*

*      Discuss *classes* versus *objects*

**III. Overview & Requirements:**

This lab, along with your TA, will help you navigate through designing, implementing, testing a List class in C++. It will also help you with understanding how to apply a List object to an application.

Labs are held in a “closed” environment such that you may ask your TA questions. Please use your TAs knowledge to your advantage. You are required to move at the pace set forth by your TA. Have a great time! Labs are a vital part to your education in CptS 122 so work diligently.

**--- Read the following before you start! ---**

A couple of notes to consider:

-       You will be working with *dynamically* allocated space in C++. Instead of using malloc(), we’ll use operator new to allocate space. Also, instead of using free() to deallocate space, we’ll use operator delete. When operator new is called, the constructor for the object is automatically invoked! When the operator delete is called, the destructor for the object is automatically invoked! Please keep these ideas in mind.

-       You will be working with *copy* constructors. You will have the option to implement a *shallow* and *deep* copy constructors. A *shallow* copy will perform a basic assignment of data members of one object to the same data members of another object. A *deep* copy will allocate extra space to ensure that the items copied are in distinct locations. A deep copy is usually considered when working with pointers! If you want to copy one list to another list, should the head pointer be copied only (shallow copy)? Or should extra space be allocated to establish a completely new list with the same data items (deep copy)?

**Tasks:**

Starting with the List code found at <https://eecs.wsu.edu/~aofallon/cpts122/labs/ClassLinkedList.zip>:

**1.**Unzip the file and review “main.cpp” with your teammates.  Answer questions 1-6, which are provided through comments in the file. Use the debugger to help answer the questions!

**2.**Review the “ListNode.h”, “ListNode.cpp”, “List.h”, and “List.cpp” files. Start to fill in the code for the functions listed below. Note: the ListNode class declaration (“ListNode.h”) and ListNode function implementations (“ListNode.cpp”) have been completed for you!

**a.**Write the implementation for the deep copy constructor the List class (found in “List.cpp”). After you finish writing your copy constructor, uncomment line 25 (List l2 = l1;) in main (). Answer questions 7 and 8.

**b.**Write the implementation for the deep copy assignment operator (found in “List.cpp”). After you finish writing your overloaded assignment operator, uncomment line 37 (l3 = l2;) in main (). Answer questions 9 and 10.

**c.**Write the implementation for insertInorder () (found in “List.cpp”). Test your function by calling it in main () or within a test function.

**d.**Write the implementation for insertAtEnd () (found in “List.cpp”). Test your function by calling it in main () or within a test function.

**e.**Write the implementation for deleteAtFront () (found in “List.cpp”). Test your function by calling it in main () or within a test function.

**f.**Write the implementation for deleteNode () (found in “List.cpp”). Test your function by calling it in main () or within a test function.

**g.**Write the implementation for deleteAtEnd () (found in “List.cpp”). Test your function by calling it in main () or within a test function.

**h.**Answer question 11 in main ().

**3.**Review the “ListApp.h” and “ListApp.cpp” files. Write an application, which computes the high, low, and mean of the scores in the “ExamScores.csv” file. The functions for reading and extracting the scores from the file, and for inserting into the list have been completed for you! See the “ListApp.h” and “ListApp.cpp” files. You will need to write functions for the computations and for writing the results to a file. You will have to visit each node in your linked list of scores to perform the computations! The high, low, and mean should be written to a file called “ExamStats.txt”. These functions should be called from the runApp () function. Note: the ListApp class constructor takes care of opening the files!

**IV. Submitting Labs:**

****You are not required to submit your lab solutions, unless you are unable to attend them synchronously. You should keep them in a folder that you may continue to access throughout the semester.

**V. Grading Guidelines:**

       This lab is worth 10 points. Your lab grade is assigned based on completeness and effort. To receive full credit for the lab you must show up on time, continue to work on the problems until the TA has dismissed you, and complete at least 2/3 of the problems.

**CptS 122 – Data Structures                                                                                  **

**Programming Assignment 8: Data Analysis using Binary Search Trees**

**Assigned:** Friday, April 9, 2021

**Due:**Friday, April 16, 2021 by midnight

**I. Learner Objectives:**

At the conclusion of this assignment, participants should be able to:

*      Analyze a basic set of requirements for implementing and testing a solution to a problem

*      Design, implement and test classes in C++

*      Design and apply inheritance

*      Design with polymorphism

*      Design and implement a dynamically linked binary search tree

**II. Prerequisites:**

Before starting this assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Compose basic C++ language programs

*      Create basic test cases for a program

*      Apply arrays, strings, and pointers

*      Declare and define *constructors*

*      Declare and define *destructors*

*      Compare and contrast *public* and *private* access specifiers in C++

*      Describe what is an *attribute* or data member of a class

*      Describe what is a *method* of a class

*      Apply and implement *overloaded* functions

*      Distinguish between pass-by-*value* and pass-by-*reference*

*      Discuss *classes* versus *objects*

**III. Overview & Requirements:**

***Summary:***

For this assignment you are implementing a system for detecting trends in consumer products over a 48-hour period. We are interested in knowing which products are purchased and sold, the least and most, by various retail stores throughout the United States. When a product is tagged as *purchased* it indicates that a certain retail store bought units of the product from a supplier. When a product is tagged as *sold* it indicates that a certain retail store sold that many units of a product. Your system must read product data from a .csv file, and store the data in a way that inserts data in better than linear time (O(n)) in most cases. Since, a binary search tree (BST) is a reasonably efficient data structure for inserting and searching data (O (log n) for balanced trees), you must create two BSTs; one tree represents the products that were sold and the other tree represents the products that were bought. The system must leverage the organization of the trees to display, which products were least bought and sold, and which were most bought and sold for that 48-hour period. Your system is only required to output the following to the screen:

-       The contents of the two BSTs, which will be printed in order

-       The product that type and number of units that sold the most

-       The product that type and number of units that sold the least

-       The product that type and number of units that were purchased the most

-       The product that type and number of units that were purchased the least

***Class Design:***

For this assignment you are required to implement a dynamically linked binary search tree. You will first need to start by defining an *abstract* base class Node, which encapsulates the following:

*Data members:*

          # mData : std::string // # denotes protected

          # mpLeft : Node \*

          # mpRight : Node \*

*Member functions:*

          + virtual destructor // + denotes public

          + constructor which accepts a string to set the data in the node; each pointer in the node is set to NULL

          + setters – one for each data member (3 total should be defined)

          + getters – one for each data member (3 total should be defined, the 2 defined to get the pointers should return a reference to the pointer, i.e. Node \*&)

          + pure virtual printData () function, which must be overridden in class TransactionNode

Next define a class TransactionNode, which *publically* inherits from abstract base class Node. Class TransactionNode must encapsulate the following:

*New Data members:*

- mUnits : int // - denotes private

*New* *Member functions:*

          + destructor // + denotes public

          + constructor which accepts a string to set the data and an integer to set the number of units in the node; should invoke class Node’s constructor

          + setter

          + getter

          + printData (), which overrides the pure virtual function in class Node

Now define a class BST, which encapsulates the following:

*Data members:*

- mpRoot : Node \* // yes, we want a pointer to a Node, not TransactionNode here!

*Member functions:*

          + destructor // calls destroyTree ()

          - destroyTree () // yes, it’s private, and it should visit each node in postOrder to delete them

          + default constructor

          + setter

          + getter

          + insert () // public used to hide pointer information, i.e. won’t pass in the root of the tree into this function, only the private insert () function

- insert () // yes, it’s private, and it dynamically allocates a TransactionNode and inserts recursively in the correct subtree based on mUnits; should pass in a reference to a pointer (i.e. Node \*& pT)

+ inOrderTraversal () // yes, once again it’s private to hide pointer information

          - inOrderTraversal (), which recursively visits and prints the contents (mData and mUnits) of each node in the tree in order; each node’s printData () should be called

  contents should be printed on a separate line; must call the printData () function associated with the TransactionNode

          + findSmallest (), which returns a reference to a TransactionNode (i.e TransactionNode &)  with the smallest mUnits

          + findLargest (), which returns a reference to a TransactionNode with the largest mUnits

Lastly, define a class DataAnalysis, which encapsulates the following:

*Data members:*

          - mTreeSold : BST

          - mTreePurchased : BST

          - mCsvStream : ifstream

*Member functions:*

- A function that opens <https://eecs.wsu.edu/~aofallon/cpts122/progassignments/data.csv> // yes, it’s private, and must use mCsvStream to open the file

          - A function that reads one line from the file and splits the line into units, type, and transaction fields

          - A function that loops until all lines are read from the file; calls the function below, and then displays the current contents of both BSTs; use inOrderTraversal () to display the trees

          - A function that compares the transaction field and inserts the units and type into the appropriate tree (mTreeSold or mTreePurchased) // note with the way the data.csv file is organized the trees will be fairly balanced

          - A function that writes to the screen the trends we see in our tree; the function must display the type and number of units that are least purchased and sold, and the most purchased and sold

          + runAnalysis (), which is the only public function in the class, aside from possibly a constructor and/or destructor; this function calls the other private functions

What should go in main ()?

          DataAnalysis obj;

          obj.runAnalysis ();

***Questions to Ponder:***

Questions for you to consider (you do not need to submit answers to these questions):

-       We understand that a BST is most efficient when it is balanced. If the data.csv file was not already organized to provide a fairly balanced tree, how would you balance the tree as you insert?

-       Do you think there are other data structures that would be better suited for this type of problem? Why?

-       What would happen to our program if we found duplicate products or # of units sold/purchased in the file? Would we need a data structure to efficiently combine the products and # of units? Hash table?

**IV. Submitting Assignments:**

1.       Using Blackboard Learn <https://learn.wsu.edu/webapps/login/> submit your assignment. You will submit your assignment in the ***lab*** Blackboard space. Under the "Content" link navigate to the "Programming Assignment Submissions" folder and upload your solution to the appropriate “Assignment” space. You must upload your solution, through an attachment, as <your last name>\_pa8.zip by the due date and time.

1. Your project must contain at least one header file (.h files) and two C++ source files (which must be .cpp files).
2. Your project must build properly. The most points an assignment can receive if it does not build properly is 65 out of 100.

**V. Grading Guidelines:**

This assignment is worth 100 points. Your assignment will be evaluated based on adherence to the requirements. We will grade according to the following criteria:

*      5 pts – Appropriate design, style, and commenting according to class standards

*      5 pts – Node class and all functions described above

*      10 pts – TransactionNode class, which inherits from the Node class, and all functions described above

*      40 pts – BST class

       5 pts - destroyTree ()

       10 pts – insert () functions: 8 pts private one, 2 points public one

       7 pts – inOrderTraversal () functions: 5 pts private one, 2 points public one

       7 pts – findSmallest ()

       7 pts – findLargest ()

       4 pts – other functions

*      35 Pts – DataAnalysis class

       2 pts - for opening data.csv

       8 pts – for reading a line and splitting it

       10 pts – for reading all lines in the file and inserting into the appropriate tree (mTreeSold and mTreePurchased)

       10 pts – for determining the trends and displaying them to the screen

       5 pts – other functions

*      5 pts – main ()

**CptS 122 – Data Structures                                                                         **

**Lab 8: Developing a Binary Search Tree (BST) Class and Application in C++**

**Assigned:** Week of March 15, 2021

**Due:**At the end of the lab session

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design, implement, test, and apply a BST class in C++

*      Compare and contrast *value* classes versus *container* classes

*      Apply and implement *overloaded* functions and operators

*      Discuss the properties of a BST

*      Discuss the advantages and disadvantages of a BST

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Create test cases for a program

*      Design, implement and test classes in C++

*      Declare and define *constructors*

*      Declare and define *destructors*

*      Compare and contrast *public* and *private* access specifiers in C++

*      Describe what is an *attribute* or data member of a class

*      Describe what is a *method* of a class

*      Apply and implement *overloaded* functions

*      Distinguish between pass-by-*value* and pass-by-*reference*

*      Discuss *classes* versus *objects*

**III. Overview & Requirements:**

This lab, along with your TA, will help you navigate through designing, implementing, and testing a BST class in C++. It will also help you with understanding how to apply a BST object to an application.

Labs are held in a “closed” environment such that you may ask your TA questions. Please use your TAs knowledge to your advantage. You are required to move at the pace set forth by your TA. Have a great time! Labs are a vital part to your education in CptS 122 so work diligently.

**Tasks:**

**NOTE:**Parts of this lab are courtesy of Jack Hagemeister.

For this lab you will develop a BST class and use it to solve a sorting problem. You will create the class BSTNode declaration in a single .h (called BSTNode.h) and the class BST declaration in a single .h file (called BST.h). Define all of the functionality for class BSTNode in BSTNode .cpp and functionality for class BST in BST.cpp.

**Task 1. Defining a class, BSTNode**

Start this task by designing the BSTNode class for the BST. For the initial development you should just build the node to hold a std::string as its data. The BSTNode class will consist of a string and left and right pointers. It will initialize the node using it’s constructor. You will also overload the stream insertion operator << to output a node. Will you need access and/or modify the contents of the nodes from outside the node? Yes! Then you should implement getters/setters. Note: you will be inserting data into the tree using recursion. How will this impact your getters in the BSTNode class? The data in the node is a std::string, should we pass the newData value into the setter using pass-by-reference? Recall, a std::string is an object type, and hence, if the object is not passed by reference, then a copy of the object is made (std::string copy constructor is invoked). Is this the intent? Probably not!

**Task 2. Now create the BST class**

Implement a BST class. You are now ready to define the BST class. You should create a data member for a pointer that will be the root of the BST. The pointers should be of type BSTNode. You will also implement the constructor and the destructor (should destroy the tree through postorder traversing of nodes).

Additionally, you need:

          insertNode() -  that adds an item to the BST. Recall the properties of a BST. The values in any left subtree are less than its parent node,

                                  and any values in the right subtree are greater than its parent node. Use recursion in your implementation!

          inOrderTraversal() - that prints the contents of the BST inorder. Use recursion in your implementation!

          preOrderTraversal() -  that prints the contents of the BST preorder. Use recursion in your implementation!

          postOrderTraversal() -  that prints the contents of the BST postorder. Use recursion in your implementation!

          isEmpty() -  that is a Boolean function that indicates that the BST is empty or not.

          destroyTree() – a private function, which is called from the destructor to delete each node in the the tree by postorder traversal.

You will overload the stream insertion operator << to output a BST in an elegant way.

NOTE: Listed below are the algorithms for the traversals.

In-Order Traversal:

1.    Traverse the “left” subtree by recursively calling inOrderTraversal()

2.    Access the “data” of the current node

3.    Traverse the “right” subtree by recursively calling inOrderTraversal()

Pre-Order Traversal:

1.    Access the “data” of the current node

2.    Traverse the “left” subtree by recursively calling preOrderTraversal()

3.    Traverse the “right” subtree by recursively calling preOrderTraversal()

Post-Order Traversal:

1.    Traverse the “left” subtree by recursively calling postOrderTraversal()

2.    Traverse the “right” subtree by recursively calling postOrderTraversal()

3.    Access the “data” of the current node

Let’s say we have the following BST:

                    42

               /         \

           25             75

         /     \         /   \

       10     30      65    100

      /  \    / \      / \     / \

         15               70

The in-order traversal would print: 10 15 25 30 42 65 70 75 100

The pre-order traversal would print: 42 25 10 15 30 75 65 70 100

The post-order traversal would print: 15 10 30 25 70 65 100 75 42 // The value in each node is not printed until the values of its children are printed

**Task 3. Create a an application to sort strings**

Create an application that populates an array with names (last, first) of your favorite people. Read the names from a text file, where each name is placed on a separate line in the form (last, first). Take the array and place all names in the array into a BST object. Traverse the BST inorder (you will need to modify inorder) and place the inorder strings back into the original array. The array of people's names is now sorted. QUESTION: When would it be a good idea to use a BST for sorting items? Do you know of other algorithms and data structures that are more efficient for a sorting task?

**IV. Submitting Labs:**

       You are not required to submit your lab solutions, unless you are unable to attend them synchronously. You should keep them in a folder that you may continue to access throughout the semester.

**V. Grading Guidelines:**

       This lab is worth 10 points. Your lab grade is assigned based on completeness and effort. To receive full credit for the lab you must show up on time, continue to work on the problems until the TA has dismissed you, and complete at least 2/3 of the problems.

CptS**122 – Data Structures                                                                                             **

**Programming Assignment 9: A Graphical Game or Application**

**Assigned:** Wednesday, April 14, 2021

**Due:**Friday, April 30, 2021 by midnight (extended date)

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design, implement and test classes in C++

*      Apply game or application design principles

*      Implement and apply inheritance and polymorphism

*      Apply graphics to a solution

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Compose basic C++ language programs

*      Describe what is *inheritance*

*      Create basic test cases for a program

*      Apply arrays, strings, and pointers

*      Declare and define *constructors*

*      Declare and define *destructors*

*      Compare and contrast *public* and *private* access specifiers in C++

*      Describe what is an *attribute* or data member of a class

*      Describe what is a *method* of a class

*      Apply and implement *overloaded* functions

*      Distinguish between pass-by-*value* and pass-by-*reference*

*      Discuss *classes* versus *objects*

**III. Overview & Requirements:**

**NOTE:** I will be grading your assignment! Please be sure to clearly identify your team members and corresponding lab sections in a readme.txt file that is added to your “Resource Files” folder in your project. Only *one* of your team members will be required to submit a solution!!! Since some of your solutions may exceed the upload limits accepted by OSBLE+, you will need to transfer your solution to my computer via a flash drive or provide me with a link to your solution online.

For this final assignment, you are required to create a solution, as a team (you may have a team of up to 4 members), to a game or graphical application of your choice! Some game possibilities are listed below:

        Chess

        Texas Hold 'em

        Battleship

        Checkers

        Others?

**However, you may NOT develop a solution to the game of Snake or Pong!!!** You must apply inheritance and polymorphism in your solution. You are also required to develop a test class and implement 5 test cases for your application.

Your goal for the assignment is build a complete graphical, and possibly networked, game or application. As a team you must ultimately decide how you will implement graphics. You have many tools and library options available to implement the graphics portion of the assignment. Some include the Unreal Engine, SFML, Qt, SDL, Allegro, DirectX, OpenGL, etc. Please be sure to also add some directions of how to play the game or use your application.

Aside from the requirements listed in the above paragraph, you are free to complete this assignment as you see fit.

Have fun with this assignment!

**IV. Submitting Assignments:**

1.       Using Blackboard Learn <https://learn.wsu.edu/webapps/login/> submit your assignment. You will submit your assignment in the ***lab*** Blackboard space. Under the "Content" link navigate to the "Programming Assignment Submissions" folder and upload your solution to the appropriate “Assignment” space. You must upload your solution, through an attachment, as <your last name>\_pa9.zip by the due date and time.

1. Your .zip file should contain a project workspace. Your project folder must have at least two header files (.h files), three C++ source files (which must be .cpp files), and project workspace. Delete the debug folder before you zip your project folders.
2. Your project must build properly. The most points an assignment can receive if it does not build properly is 200 out of 300.

**V. Grading Guidelines:**

This assignment is worth 300 points. Your assignment will be evaluated based on a successful compilation and adherence to the program requirements. We will grade according to the following criteria:

*      25 pts – Appropriate design, style, and commenting according to class standards

*      75 pts – Design and implementation of main game play or application (along with user directions of how to play). **You must apply inheritance and polymorphism!**

*      100 pts – Implementation of graphics

*      25 pts - Creativity and originality in implementation of the game/application

*      25 pts – 5 test cases

*      50 pts – Demo (this will not be required for this semester)

*      50 pts BONUS - Implementation of network communication

**CptS 122 – Data Structures                                                                         **

**Lab 9: Developing a Stack Class Template in C++**

**Assigned:** Week of March 22, 2021

**Due:**At the end of the lab session

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design, implement and test a Stack class template in C++

*      Apply an array implementation for a Stack

*      Compare and contrast value classes versus container classes

*      Apply and implement *overloaded* functions and operators

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Create test cases for a program

*      Design, implement and test classes in C++

*      Declare and define *constructors*

*      Declare and define *destructors*

*      Compare and contrast *public* and *private* access specifiers in C++

*      Describe what is an *attribute* or data member of a class

*      Describe what is a *method* of a class

*      Apply and implement *overloaded* functions

*      Distinguish between pass-by-*value* and pass-by-*reference*

*      Discuss *classes* versus *objects*

**III. Overview & Requirements:**

This lab, along with your TA, will help you navigate through designing, implementing, and testing a Stack class template in C++. It will also help you with understanding how to apply a stack object to an application.

Labs are held in a “closed” environment such that you may ask your TA questions. Please use your TAs knowledge to your advantage. You are required to move at the pace set forth by your TA. Have a great time! Labs are a vital part to your education in CptS 122 so work diligently.

**Tasks:**

**NOTE:**Parts of this lab are courtesy of Jack Hagemeister.

For this lab you will develop a Stack class template to evaluate postfix expressions. You will start with the provided code.

**Tasks:**

**Recall with Big-O we want..**

–       **to determine central unit of work by considering the operations applied in the algorithm**

–       **to express unit of work as function of size of input data: How quickly does amount of work grow as size of input grows?**

–       **classify algorithms according to how their running *time* and/or *space* requirements grow as input size grows**

–       **the analysis of an algorithms is extremely useful when comparing algorithms that solve the same problem!**

**0.    Revisit the following questions from lab 4, however provide a Big-O time and space analysis for each!**

a.    Is it more efficient to delete the last node in an array or linked implementation of a list?

b.   Is it more efficient to delete the first node in an array or linked implementation of a list?

c.    Is it more efficient to delete a node, in general, in an array or linked implementation of a list?

d.   Is it more efficient to insert a node at the end in an array or linked implementation of a list?

e.   Is it more efficient to insert a node at the front in an array or linked implementation of a list?

f.    Is it more efficient to insert a node, in general, in an array or linked implementation of a list?

g.    Is it more efficient to access a node, in general, in an array or linked implementation of a list?

**Recall, a class template enables for generic programming. The template provides a stencil or pattern for a class. We can provide different specializations from the template. One way to view this is that the template is the like the stencil from which shapes are traced. Each shape traced could be specialized with different colors, textures, etc.**

Starting with the Stack template code found at <https://eecs.wsu.edu/~aofallon/cpts122/labs/StackTemplateLab.zip>:

**1.    Review Stack Class Template Code in C++.**

First, define what is a stack? What is the purpose of using stacks?

Next, review and discuss the skeleton code with your teammates. Notice that all of the template code is located only in the header (.h) files, sometimes you will see the use of (.hpp) files instead of (.h)! If you place the template code in .cpp files, you will encounter linker errors! The code currently compiles. However, some of the functions are considered “stubs” because they return values, but have not been completely implemented. You will write the code for the “stubs” in task 2. Are there any functions missing from the stack implementation? You may not know until you start implementing the functions for tasks 3 and 4! Feel free to add more functions as you see fit!

**2.    Implement Stack Code in C++.**

Second, start to fill in the code for each of the “stub” functions (found in the Stack class only) and any other functions that you have added to the Stack. The functions that you must implement include:

*      isEmpty () – what is the Big-O of this algorithm?

*      push () - what is the Big-O of this algorithm?

*      pop () - what is the Big-O of this algorithm?

*      peek () - what is the Big-O of this algorithm?

As you complete a function, test it. As part of the project you will notice that a test class has been provided. Please use this class to test your functions!

**3.    Postfix Evaluation with a Stack.**

Build a function that will complete a postfix evaluation of a given input string using your stack implementation. What is the type of item that will be stored in your stack? An integer for this part! Note: each operand is a *single* digit only. The function should return the value of the expression. Test your implementation on the following inputs; you can assume that the inputs are valid.

6  2  +  5  \*  8  4  /  -

5  6  2 \*  + 9  /

          4  5  /  6  8  \*  -

***Algorithm for Evaluating Postfix Expressions***

-       **What is the Big-O of the following algorithm?**

1.    Let **S** be an empty stack.

2.    If there is no character to read, then the postfix expression is malformed.  This error in the input string should be reported and the program should give up on that string.

3.    Read the next character and call it **c**.

4.    If **c** is the symbol '='

1.    If **S** is empty, then the postfix expression is malformed.  This error in the input string should be reported and the program should give up on that string.

2.    If **S** has more than one element, then the postfix expression is malformed.  This error in the input string should be reported and the program should give up on that string.

3.    If **S** has exactly one element, call it **e**.  The value of the postfix expression is **e**.  Return **e**.

5.    If **c** is a digit, then push **c** onto **S** and go to Step 2.

6.    If **c** is a binary operator, call it **o**

1.    If **S** does not have at least two elements, then the postfix expression is malformed.  This error in the input string should be reported and the program should give up on that string.

2.    Otherwise, pop two elements off of **S**.  Call the first one popped **s2** and the second one popped **s1**.  I.e., **s1** was below **s2** in **S** prior to the popping.

3.    Apply the operator, **o**, to **s1** and **s2** (in that order) to obtain a value called **v**.

4.    push **v** onto **S**.

5.    Go to Step 2.

**4.    Convert infix to postfix.**

-       **What is the Big-O of the following algorithm?**

Using your stack class template implementation, create a function that will convert a given infix expression (string) to a postfix expression. In this case the stack must contain character items (representative of the operator)! Recall, each operand is a *single* digit only.

See the page <https://eecs.wsu.edu/~aofallon/cpts122/labs/Infix2Postfix.pdf> for an explanation and algorithm.

**IV. Submitting Labs:**

       You are not required to submit your lab solutions, unless you are unable to attend them synchronously. You should keep them in a folder that you may continue to access throughout the semester.

**V. Grading Guidelines:**

       This lab is worth 10 points. Your lab grade is assigned based on completeness and effort. To receive full credit for the lab you must show up on time, continue to work on the problems until the TA has dismissed you, and complete at least 2/3 of the problems.

**CptS 122 – Data Structures                                                                         **

**Lab 10: Big-O and Merge Sort in C++**

**Assigned:** Week of March 29, 2021

**Due:**At the end of the lab session

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Implement the merge sort algorithm in C++

*      Analyze the best-case, average-case, and worst-case scenarios for runtime of merge sort

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Define Order of Magnitude (Big-O)

*      Analyze algorithms and determine the best-case, average-case, and worst-case scenarios using Big-O

**III. Overview & Requirements:**

Labs are held in a “closed” environment such that you may ask your TA questions. Please use your TAs knowledge to your advantage. You are required to move at the pace set forth by your TA. Have a great time! Labs are a vital part to your education in CptS 122 so work diligently.

**Tasks:**

**Recall (again ) with Big-O we want..**

–       **to determine central unit of work by considering the operations applied in the algorithm**

–       **to express unit of work as function of size of input data: How quickly does amount of work grow as size of input grows?**

–       **classify algorithms according to how their running *time* and/or *space* requirements grow as input size grows**

–       **the analysis of an algorithm is extremely useful when comparing algorithms that solve the same problem!**

0.    If you were not able to address the questions about Big-O last lab, please revisit the following questions. Provide a Big-O time and space analysis for each!

a.    Is it more efficient to delete the last node in an array or linked implementation of a list?

b.   Is it more efficient to delete the first node in an array or linked implementation of a list?

c.    Is it more efficient to delete a node, in general, in an array or linked implementation of a list?

d.   Is it more efficient to insert a node at the end in an array or linked implementation of a list?

e.   Is it more efficient to insert a node at the front in an array or linked implementation of a list?

f.    Is it more efficient to insert a node, in general, in an array or linked implementation of a list?

g.    Is it more efficient to access a node, in general, in an array or linked implementation of a list?

1.    Implement a C++ solution to merge sort. First, view the video at: <https://www.youtube.com/watch?v=IR7DFDej6l4>. Second, discuss in English an algorithm to describe the steps shown in the video. Third, implement a function template for your solution in C++. Once you’ve completed your solution analyze the best-case, average-case, and worst-case scenarios for merge sort.

2.    Work on the current programming assignment.

**IV. Submitting Labs:**

*      You are not required to submit your lab solutions, unless you are unable to attend them synchronously. You should keep them in a folder that you may continue to access throughout the semester.

**V. Grading Guidelines:**

*      This lab is worth 10 points. Your lab grade is assigned based on completeness and effort. To receive full credit for the lab you must show up on time, continue to work on the problems until the TA has dismissed you, and complete at least 2/3 of the problems.

**CptS 122 – Data Structures                                                                         **

**Lab 11: Inheritance in C++**

**Assigned:** Week of April 5, 2021

**Due:**At the end of the lab session

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design, implement and test classes in C++ which apply inheritance

*      Compare and contrast inheritance (“is-a”) relationships versus composition (“has-a”) relationships

*      Apply and implement *overloaded* functions and operators

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Create test cases for a program

*      Design, implement and test classes in C++

*      Declare and define *constructors*

*      Declare and define *destructors*

*      Compare and contrast *public, protected,* and *private* access specifiers in C++

*      Describe what is an *attribute* or data member of a class

*      Describe what is a *method* of a class

*      Apply and implement *overloaded* functions

*      Distinguish between pass-by-*value* and pass-by-*reference*

*      Discuss *classes* versus *objects*

**III. Overview & Requirements:**

This lab, along with your TA, will help you navigate through designing, implementing, and testing inheritance with classes in C++. It will also help you with understanding how to apply inheritance to an application. You should NOT apply polymorphism in this lab!

Labs are held in a “closed” environment such that you may ask your TA questions. Please use your TAs knowledge to your advantage. You are required to move at the pace set forth by your TA. Please help other students in need when you are finished with a task. You may work in pairs if you wish. However, I encourage you to compose your own solution to each problem. Have a great time! Labs are a vital part to your education in CptS 122 so work diligently.

**I highly encourage you to work in teams for these tasks!**

**Tasks:**

**Task 1:**Create a base *class* called *Person*, which has *private* data members for a name, age, gender, and height. Implement a constructor, copy constructor, destructor, overloaded assignment operator, overloaded stream insertion and extraction operators, and getters and setters. See Task 2 for testing this class.

**Task 2:**Create a derived *class* called *TestPerson*, which *publically* inherits from *class* *Person*. Create test methods for each method in *class* *Person*. Remember these functions should not accept any parameters or return any values. However, they should print messages for “test case passed” or “test case failed”.

**Task 3:**Modify your *class* *Person* so that the data members are *protected* instead of *private*. How does this affect the tests cases that you created in Task 2?

**Task 4:** Create a derived *class* called *Student*, which *publically* inherits from *class* Person. Add three *private* data members to *class* *Student*. These include an array of *struct* *Course*, the number of courses taken, and the total number of credits. The *struct* *Course* should contain a string for course name, credits assigned to course, and current grade in course. Implement appropriate constructors, overloaded operators, and setters and getters for this class. Also, implement methods for computing total credits taken and current GPA. How would your implementation change if you decided to define a *class* *Course* instead of *struct* *Course*?

**Task 5:** Create a derived *class* called *Teacher*, which *publically* inherits from *class* *Person*. Add three *private* data members to *class* *Teacher*. These include an array of *struct* *Course*, the number of courses taken, and the total number of credits. The *struct* *Course* should contain a string for course name, credits assigned to course, and average grade of students’ in course. Implement appropriate constructors, overloaded operators, and setters and getters for this class. Also, implement methods for computing total credits taught and average grades of students across the courses taught. How would your implementation change if you decided to derive *class* *Teacher* from *class* *Student* instead of *class* *Person*?

**Task 6:** Create an application, which allows students to register for classes taught by a particular teacher.

**IV. Submitting Labs:**

*      You are not required to submit your lab solutions, unless you are unable to attend them synchronously. You should keep them in a folder that you may continue to access throughout the semester.

**V. Grading Guidelines:**

*      This lab is worth 10 points. Your lab grade is assigned based on completeness and effort. To receive full credit for the lab you must show up on time, continue to work on the problems until the TA has dismissed you, and complete at least 2/3 of the problems.

**CptS 122 – Data Structures                                                                         **

**Lab 12: Inheritance and Containers in C++**

**Assigned:** Week of April 12, 2021

**Due:**At the end of the lab session

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design, implement and test classes in C++ which apply inheritance

*      Apply and implement private inheritance to container classes

*      Compare and contrast inheritance (“is-a”) relationships versus composition (“has-a”) relationships

*      Apply and implement *overloaded* functions and operators

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Create test cases for a program

*      Design, implement and test classes in C++

*      Declare and define *constructors*

*      Declare and define *destructors*

*      Compare and contrast *public* and *private* access specifiers in C++

*      Describe what is an *attribute* or data member of a class

*      Describe what is a *method* of a class

*      Apply and implement *overloaded* functions

*      Distinguish between pass-by-*value* and pass-by-*reference*

*      Discuss *classes* versus *objects*

*      Describe and define *inheritance*

**III. Overview & Requirements:**

This lab, along with your TA, will help you navigate through designing, implementing, and testing inheritance with container classes in C++. It will also, once again, help you with understanding how to apply *inheritance* to an application.

Labs are held in a “closed” environment such that you may ask your TA questions. Please use your TAs knowledge to your advantage. You are required to move at the pace set forth by your TA. Please help other students in need when you are finished with a task. You may work in pairs if you wish. However, I encourage you to compose your own solution to each problem. Have a great time! Labs are a vital part to your education in CptS 122 so work diligently.

**Tasks:**

**NOTE:**Parts of this lab are courtesy of Jack Hagemeister.

**One of the powers of inheritance is that it facilitates large amounts of code reuse. In this lab you will redesign your queue class by inheriting from a base list class.**

**Task 1. List**

Implement a templated class list and listnode. You may add methods/functions as you see fit. Test these classes. I have left all of the implementation as an exercise for you.

**template<** **class** NODETYPE **>** **class** List**;**  **// forward declaration**

**template<class** NODETYPE**>**

**class** ListNode

**{**

**friend** **class** List**<** NODETYPE **>;** **// make List a friend**

**public:**

   ListNode**(** **const** NODETYPE **&**newData**);**  **// copy constructor**

   NODETYPE getData**()** **const;**      **// return data in the node**

**private:**

   NODETYPE data**;**                 **// data**

   ListNode**<** NODETYPE **>** **\***nextPtr**;** **// next node in the list**

**};**

**template<** **class** NODETYPE **>**

**class** List

**{**

**public:**

   List**();**      **// constructor**

**~**List**();**     **// destructor**

**void** insertAtFront**(** **const** NODETYPE **&**newData**);**

**void** insertAtBack**(** **const** NODETYPE **&**newData **);**

**bool** removeFromFront**(** NODETYPE **&**removedData **);**

**bool** removeFromBack**(** NODETYPE **&**removedData **);**

**bool** isEmpty**()** **const;**

**void** print**()** **const;**

**private:**

   ListNode**<** NODETYPE **>** **\***firstPtr**;**  **// pointer to first node**

   ListNode**<** NODETYPE **>** **\***lastPtr**;**   **// pointer to last node**

**// Utility function to allocate a new node**

   ListNode**<** NODETYPE **>** **\***getNewNode**(** **const** NODETYPE **&**newData **);**

**};**

**Task 2. Queue**

Create a Queue class template that *privately* inherits from a List class.  You should define enqueue ( ) and dequeue ( ) operations in terms of the inherited list operations.

**Task 3. Network Traffic Application**

Write an application that simulates network traffic. The traffic is represented by packets (of information) moving through the network. These packets must be represented by a class. Each packet must include an integer length field (in bytes) and a std::string field for data, where the length field is the number of characters in the std::string. You must represent one device in the network, which is represented by a Queue object. The application must randomly assign the arrival time of the first packet and the time that it takes to process the packet at the device. As a new packet arrives to the device, the arrival time for the next packet should be generated. Every time a packet leaves the device or a new packet arrives, print out the packet information for the one at the front and the back of the Queue.

**IV. Submitting Labs:**

*      You are not required to submit your lab solutions, unless you are unable to attend them synchronously. You should keep them in a folder that you may continue to access throughout the semester.

**V. Grading Guidelines:**

*      This lab is worth 10 points. Your lab grade is assigned based on completeness and effort. To receive full credit for the lab you must show up on time, continue to work on the problems until the TA has dismissed you, and complete at least 2/3 of the problems.

**CptS 122 – Data Structures                                                                         **

**Lab 13: Polymorphism in C++**

**Assigned:** Week of April 19, 2021

**Due:**At the end of the lab session

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design, implement and test classes in C++ which apply polymorphism

*      Apply inheritance and polymorphism to model and simulate Animals and Pets

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Create test cases for a program

*      Design, implement and test classes in C++ which apply inheritance

*      Compare and contrast inheritance (“is-a”) relationships versus composition (“has-a”) relationships

*      Declare and define *constructors*

*      Declare and define *destructors*

*      Compare and contrast *public* and *private* access specifiers in C++

*      Describe what is an *attribute* or data member of a class

*      Describe what is a *method* of a class

*      Apply and implement *overloaded* functions

*      Distinguish between pass-by-*value* and pass-by-*reference*

*      Discuss *classes* versus *objects*

*      Describe and define *inheritance*

**III. Overview & Requirements:**

This lab, along with your TA, will help you navigate through designing, implementing, and testing polymorphism with classes in C++. It will also, once again, help you with understanding how to apply inheritance to an application.

Labs are held in a “closed” environment such that you may ask your TA questions. Please use your TAs knowledge to your advantage. You are required to move at the pace set forth by your TA. Have a great time! Labs are a vital part to your education in CptS 122 so work diligently.

**Tasks:**

**NOTE:**Parts of this lab are courtesy of Jack Hagemeister.

**Please work in your teams for this lab!!!!**

**Task 1.**

To gain a better understanding of polymorphic and virtual functions start with the following simple example. Notice we have not defined a virtual function yet.

// Task1.h

#include <iostream>

using std::cout;

using std::endl;

class Base

{

      public:

            void testFunction ();

};

class Derived : public Base

{

      public:

            void testFunction ();

};

// Task1.cpp

#include "Task1.h"

void Base::testFunction ()

{

    cout << "Base class" << endl;

}

void Derived::testFunction ()

{

    cout << "Derived class" << endl;

}

// main.cpp

#include "Task1.h"

int main(void)

{

    Base\* ptr = new Base;

    ptr -> testFunction ();         // prints "Base class"

    delete ptr;

    ptr = new Derived;

    ptr -> testFunction ();         // prints "Base class" because the base class function is not virtual

    delete ptr;

    return 0;

}

Now modify the code with the following (all other code should remain the same).

class Base

{

      public:

            virtual void testFunction ();

};

Compile and run your program with this modification. You’ll notice the second testFunction() call generates the message “Derived class”. Welcome to polymorphism!

**Task 2.**

You will first build two classes, Mammal and Dog. Dog will inherit from Mammal. Below is the Mammal class code. Once you have the Mammal class built, build a second class Dog that will inherit publicly from Mammal.

// Mammal.h

#pragma once

#include <iostream>

using std::cout;

using std::endl;

class Mammal

{

      public:

            Mammal(void);

            ~Mammal(void);

            virtual void Move() const;

            virtual void Speak() const;

      protected:

            int itsAge;

};

// Mammal.cpp

#include "Mammal.h"

Mammal::Mammal(void):itsAge(1)

{

      cout << "Mammal constructor..." << endl;

}

Mammal::~Mammal(void)

{

      cout << "Mammal destructor..." << endl;

}

void Mammal::Move() const

{

      cout << "Mammal moves a step!" << endl;

}

void Mammal::Speak() const

{

      cout << "What does a mammal speak? Mammilian!" << endl;

}

Once you have completed class Mammal and Dog, build the following main program.

#include "Mammal.h"

#include "Dog.h"

int main ()

{

   Mammal \*pDog = new Dog;

   pDog->Move();

   pDog->Speak();

   //Dog \*pDog2 = new Dog;

   //pDog2->Move();

   //pDog2->Speak();

   return 0;

}

What does it output, is that what you expected?  Remove the keyword virtual from the class mammal and try it again. Now what happens? Next, put in another pointer to pDog2 in the main program, but this time make it a pointer to a Dog, not a mammal and create a new dog. Now what happens? What you should realize is that by making the method Speak virtual, we can have a little different behavior through dynamic (runtime) binding.

**Task 3.**

Develop additional classes for Cat, Horse, and GuineaPig overriding the move and speak methods. (If you do not know guinea pigs go “weep weep”)

Next, test with the modified main:

**int** main **()**

**{**

   Mammal**\*** theArray**[**5**];**

   Mammal**\*** ptr**;**

**int** choice**,** i**;**

**for** **(**i **=** 0**;** i**<**5**;** i**++)**

**{**

      cout **<<** "(1)dog (2)cat (3)horse (4)guinea pig: "**;**

      cin **>>** choice**;**

**switch** **(**choice**)**

**{**

**case** 1**:** ptr **=** **new** Dog**;**

**break;**

**case** 2**:** ptr **=** **new** Cat**;**

**break;**

**case** 3**:** ptr **=** **new** Horse**;**

**break;**

**case** 4**:** ptr **=** **new** GuineaPig**;**

**break;**

**default:** ptr **=** **new** Mammal**;**

**break;**

**}**

      theArray**[**i**]** **=** ptr**;**

**}**

**for** **(**i**=**0**;**i**<**5**;**i**++)**

      theArray**[**i**]->**Speak**();**

**return** 0**;**

}

**Some things to note:**

If the Dog object had a method, WagTail(), which is not in the Mammal, you could not use the pointer to Mammal to access that method (unless you cast it to be a pointer to Dog). Because WagTail() is not a virtual function, and because it is not in a Mammal object, you can't get there without either a Dog object or a Dog pointer to the Dog object!!!

The virtual function magic (polymorphic behavior) operates only on pointers and references. Passing an object by value will not enable the virtual functions to be invoked.

**Some questions that you should understand:**

Are inherited members and functions passed along to subsequent generations? If Dog derives from Mammal, and Mammal derives from Animal, does Dog inherit Animal's functions and data?

A. Yes. As derivation continues, derived classes inherit the sum of all the functions and data in all their base classes.

Q. If, in the example above, Mammal overrides a function in Animal, which does Dog get, the original or the overridden function?

A. If Dog inherits from Mammal, it gets the function in the state Mammal has it: the overridden function.

Q. Can a derived class make a public base function private?

A. Yes, and it remains private for all subsequent derivations.

Q. Why not make all class functions virtual?

A. There is overhead with the first virtual function in the creation of a v-table. After that, the overhead is trivial. Many C++ programmers feel that if one function is virtual, all others should be. Other programmers disagree, feeling that there should always be a reason for what you do.

Q. If a function (SomeFunc()) is virtual in a base class and is also overloaded, so as to take either an integer or two integers, and the derived class overrides the form taking one integer, what is called when a pointer to a derived object calls the two-integer form?

A. The overriding of the one-int form hides the entire base class function, and thus you will get a compile error complaining that that function requires only one int.

**Here are some more questions:**

1. What is a v-table?

2. What is a virtual destructor?

3. How do you show the declaration of a virtual constructor?

4. How can you create a virtual copy constructor?

5. How do you invoke a base member function from a derived class in which you've overridden that function?

6. How do you invoke a base member function from a derived class in which you have not overridden that function?

7. If a base class declares a function to be virtual, and a derived class does not use the term virtual when overriding that class, is it still virtual when inherited by a third-generation class?

8. What is the protected keyword used for?

**Some more exercises:**

1. Show the declaration of a virtual function that takes an integer parameter and returns void.

2. Show the declaration of a class Square, which derives from Rectangle, which in turn derives from Shape.

3. If, in Exercise 2, Shape takes no parameters, Rectangle takes two (length and width), but Square takes only one (length), show the constructor initialization for Square.

4. Write a virtual copy constructor for the class Square (in Exercise 3).

5. BUG BUSTERS: What is wrong with this code snippet?

void SomeFunction (Shape);

Shape \* pRect = new Rectangle;

SomeFunction(\*pRect);

6. BUG BUSTERS: What is wrong with this code snippet?

class Shape() { public: Shape(); virtual ~Shape(); virtual Shape(const Shape&); };

**IV. Submitting Labs:**

*      You are not required to submit your lab solutions, unless you are unable to attend them synchronously. You should keep them in a folder that you may continue to access throughout the semester.

**V. Grading Guidelines:**

*      This lab is worth 10 points. Your lab grade is assigned based on completeness and effort. To receive full credit for the lab you must show up on time, continue to work on the problems until the TA has dismissed you, and complete at least 2/3 of the problems.

CptS**122 – Data Structures                                                                                             **

**Bonus Assignment: Building your Own Compiler in C++**

**Due:**Friday, May 7 by midnight; please send solution to [aofallon@wsu.edu](mailto:aofallon@wsu.edu)

**I. Learner Objectives:**

At the conclusion of this programming assignment, participants should be able to:

*      Design, implement and test classes in C++

*      Apply and implement *overloaded* operators and functions

*      Maintain and expand on a previous C++ solution

**II. Prerequisites:**

Before starting this programming assignment, participants should be able to:

*      Analyze a basic set of requirements for a problem

*      Compose basic C++ language programs

*      Create basic test cases for a program

*      Apply arrays, strings, and pointers

*      Declare and define *constructors*

*      Declare and define *destructors*

*      Compare and contrast *public* and *private* access specifiers in C++

*      Describe what is an *attribute* or data member of a class

*      Describe what is a *method* of a class

*      Apply and implement *overloaded* functions

*      Distinguish between pass-by-*value* and pass-by-*reference*

*      Discuss *classes* versus *objects*

**III. Overview & Requirements:**

**NOTE:** Parts of this assignment are courtesy of Jack Hagemeister.

Preview the description for “Building Your Own Compiler” at the following site: <http://deitel.com/bookresources/cpphtp10/CompilerExercise.pdf>. You should first complete the Simpletron Computer simulator before starting this project (problems 8.15 – 8.17). Some of you may have experimented with the traditional Basic programming language. This project involves building a compiler / assembler for a restricted version of Basic called Simple. You are required to design and implement the compiler and assembler that is described in the listed .pdf.  You are required to complete exercises 19.29 – 19.33. You will have to change the design from that described to have a multiple steps to the compilation process. These are described in the Phases below.

At the completion of this project, you will have developed a complete environment for developing and running Simpletron programs. You may even have an interface that will allow a user to write and edit simple programs or simple assembly programs, compile/assemble them, and then run them in a Simpletron Simulator.

**Phase I:**

For this first phase, you will want to design value or container type objects that will provide the functionality to convert Simpletron assembly programs to Simpletron machine programs. You will need to lay out the ending machine code so that the executable statements start at location 0000 in memory. Any variables or constant values that are then used by the program should be laid out in high memory, e.g. starting from 9999 and working down.

See Chapter 8 for examples of Simpletron assembly programs and the resulting machine code. Your assembly should support all of the operations that are available in the Simpletron simulator that you have completed.

**Phase II:**

The second phase is building a compiler/translator from Simple to Simpletron assembly. You should end up with a description of a Simpletron assembly program just like those in Chapter 19. By this point you should be tired of writing them by hand. Your goal is to again design objects that can be dynamically allocated to let you use them if you so choose.

**Phase III:**

Putting it all together! When you have a working translator and simulation programs, you will want to integrate this all into one program. You will want to be able to load and save programs, as well as edit and compile them. The final part is the integration of the Simpletron simulator that lets you run the programs and step through them to see what is generated by the “SIMPLE” microcomputer.

**IV. Submitting Assignments:**

1. Send your solution to [aofallon@wsu.edu](mailto:aofallon@wsu.edu).
2. Your .zip file should contain a project workspace. Your project folder must have at least one header file (a .h file), two C++ source files (which must be .cpp files), and project workspace. Delete the debug folder before you zip your project folders.
3. Your project must build properly. You could earn up to 3% of your overall grade.

**V. Grading Guidelines:**

This assignment is worth 200 points. Your assignment will be evaluated based on a successful compilation and adherence to the program requirements. We will grade according to the following criteria:

*      10 pts – Appropriate design, style, and commenting according to class standards

*      40 pts – Design and implement the Simple assembler class (Phase I)

*      50 pts – Design and implement the Simple compiler tool (Phase II)

*      50 pts – Design and implement text-based wrapper for the compiler to assembly (Phase II) part. You should also run the assembly programs through your Phase I tool to check functionality.

*      50 pts – Develop an application to illustrate a running compiler and Simpletron (Phase III).