Bubba Hotep Moving and Storage, Inc. (BHMSI) – *Linked-List Version*

Background

The latest version of our software has been a great success. For now, we are free from the need to release another version. However, engineering is on notice that, among other enhancements, we must use a linked list, instead of a vector, to store the `MoveOrder` objects. We will begin immediately to make that modification. This project does not need to be a complete system for release. Therefore, we will do away with some of the Project #4 features. This will enable us to focus more closely on the linked list and memory management. The menu and some other functions are omitted. The file format and data validation requirements are unchanged.

Software Requirements Overview

With the exception of the menu and reports that are removed, most of the other software requirements are the same as for Project #3 and Project #4. Since there is no menu, the full path and name of the input data file shall be passed to the application as a command line argument. A command line argument is anything typed in the terminal window after typing the name of the executable file and before pressing the enter key. If you are developing your projects using an Integrated Development Environment (IDE), there should be a way to "simulate" passing command line arguments to the application. See me, or one of the teaching assistants, if you need help with that setting. Your application must test to determine how many command line arguments were actually passed. If there are less than two (the executable name itself counts as one), output an error message and do not perform any other processing. Any command line argument (after the first) shall be assumed to be the full path and name of an input data file. Your program shall open and process the contents of the input data file passed as a command line argument. Processing means to open the input data file and associate it with a stream object. If the file opens successfully, then all rows shall be read, used to update the data members of a `MoveOrder` object, and subsequently added to a linked list of `MoveOrder` objects.

File Processing

The file format is unchanged from Project #3. We will not know how many total records are in the file. Your software shall continue reading and processing lines of data until reaching the end of the input file. We have overloaded the stream extraction operator for the `MoveOrder` class. This provides the opportunity to, in one concise expression, extract an entire row of data from the file and load those values into the data members of a `MoveOrder` object. A linked list data structure shall replace the vector that we used in Project #4. All `MoveOrder` objects shall be appended to the linked-list of `MoveOrder` objects, even if errors exist in that row of data. As each row of data is processes, we shall call the `printWithMessages()` method of the `MoveOrder` object to display that row of data on the terminal.

Data Validation (no change from previous projects)

Calculations (no change from previous projects)

Functions

The following functions from Project #4 have been removed:

```
char displayMenu();
void orderDetails(const vector<MoveOrder> &);
void allDetails(const vector<MoveOrder> &);
```

All other functions have been modified to work with a linked list instead of a vector. Otherwise, each function has the same purpose as it did in Project #4. The modified function prototypes are shown below along with a brief description of the function's purpose. There is also a new function (`clearLL`) that was not part of Project #4.

```
void loadFile(string fName, bool &loadSuccess, unsigned long &count, MoveOrder* &head);
```

The parameters for this function are the input data file name and path, a `bool` variable to store `true` if the file is successfully loaded, an `unsigned long` to store the number of objects loaded, and a pointer to a MoveOrder object. Function `loadFile` will change the values stored in `loadSuccess`, `count`, and `head` so each of those variables is passed by reference. Similar to Project #4, `loadSuccess` should be set to `true` if the number of objects on the linked list is larger when the function ends than it was when the function first began. The variable `count` shall be incremented each time another object is added to the linked list. The variable `head` represents one of the fundamental differences between this project and Project #4. We are no longer storing objects on a vector. Instead, we shall use a linked list data structure to store our list of `MoveOrder` objects. In the same way that vectors increase in size at runtime, our linked list must also increase in size, as necessary, while the program is running. The variable `head` is a pointer to a `MoveOrder` object. Specifically, `head` points to the first `MoveOrder` object on the linked list. Function `loadFile` is largely unchanged from Project #4, except that it adds `MoveOrder` objects to the linked list pointed to by `head` (instead of pushing objects onto a vector). This requires dynamically allocating memory at runtime for each object that is added to the linked list. Your code may append objects to the end of the linked list or insert objects at the front of the list.

```
void clearLL( unsigned long &count, MoveOrder* &head );
```

This new function is critical to Project #5 as it has the task of deallocating all memory that was dynamically allocated for the linked list of `MoveOrder` objects. Your implementation code should traverse the linked list pointed to by head to deallocate memory for each object on the list. As each object is deleted, the `count` variable shall be decremented. When the function ends, assuming it executed completely and correctly, all objects that were on the linked list will have been deleted, head will have been updated to point to `NULL`, and the `count` variable will have been decremented until its value is zero.

```
void summaryByType( unsigned long count, MoveOrder* head );

void summaryByRegion( unsigned long count, MoveOrder* head );
```

The two summary report functions are unchanged from Project #4, except that they must traverse the linked list of `MoveOrder` objects pointed to by `head` instead of iterating through the elements of a vector.

Demonstrating the Functionality of Your Code

To demonstrate the functionality of your program, add code to function main that calls the `loadFile` function. Then call the `summaryByType` function and the `summaryByRegion` function. Finally call the `clearLL` function. The first function call results in a linked list of `MoveOrder` objects. The report functions will traverse the linked list and produce output that should match the sample program. And, finally, the `clearLL` function ensures that the objects on the list are correctly deallocated leaving no memory leaks or dangling pointers.

Grade Rubric (to be published separately)

How to approach this program

You are strongly encouraged to complete some form of design before starting to write code. Several milestones are provided for the code writing portion of this project. You are NOT required to turn anything in or to meet these milestones. I believe not meeting these milestones will put you at risk of missing the submission deadline. At each milestone, make sure that your code compiles and runs prior to moving on to the next milestone.

Milestone 1 – NLT April 11th

- Study the project description and prepare questions to discuss during class
- Create an empty source code file; insert heading comments (copy and paste from Canvas, edit as applicable), add preprocessor directives, add `using namespace std;`
- Add a "skeleton" of function `main()`
- Add global constants from Project #4
- Add the modified `MoveOrder` class declaration
- Copy and paste your implementation code from Project #4 for the `MoveOrder` class member functions, edit the heading of the constructor with parameters, there is one additional parameter for P5, otherwise, the code should compile (you do not need to implement the destructor at this time, **but you must at least add a function stub**)
- Compile and run (nothing much should happen; this is just a check for compiler errors)
- Copy and paste the provided code for the `MoveOrder` class destructor
- Copy and paste the code for Milestone 1, Test Suite 1 into your function `main()`, run the test
- Note: when running milestone tests, comment out any other code you have in function `main` except for the `return 0;` statement at the very end

Milestone 2 – NLT April 14th

- Modify your project to use command line arguments, add the test for the proper number of command line arguments as described previously on page 1 of this document
- If you are using an IDE edit your project such that one of the input data files from Project #4 is passed as a command line argument
- If you are using the server for development, then you actually type the file name on the command line after the executable file name and before pressing enter
- Remove test code for Milestone 1, Test Suite 1
- Make your code in `main` a comment, the test code goes between your (commented out) code and `return 0;`
- Copy and paste the code for Milestone 2, Test Suite 1 into your function `main()`, run the test

Milestone 3 – NLT April 23rd

- Remove the code for Milestone 2, Test Suite 1
- Add implementation code for `MoveOrder` class member functions `getNext` and `setNext`
- Add the pointer variable `MoveOrder *head = NULL` to function `main()` (this pointer will keep track of the linked list that is replacing the vector from Project #4)
- Add an `unsigned long count = 0` variable to function main to store the size of the linked list (that is, the number of `MoveOrder` objects that are on the linked list, this replaces the functionality of the `size()` function that was provided by the vector class in Project #4)
- Add the prototype for function `clearLL` and write your implementation code for this new function
- Make your code in `main` a comment, the test code goes between your (commented out) code and `return 0;`
- Copy and paste the code for Milestone 3, Test Suite 1 into your function `main()`, run the test

Milestone 4 – April 25th

- Remove the code for Milestone 3, Test Suite 1
- Add the modified prototype for the `loadFile` function
- Copy and paste your implementation code from Project #4 for the `loadFile` function
- Modify the `loadFile` function (this requires some serious thought and planning, **do not begin coding until you have a good plan of what to change**, below are a few general changes that are required)
    - change the heading to match the parameter list of the prototype
    - change the function implementation code to use a linked list instead of a vector
    - change the function implementation code to dynamically allocate memory for each `MoveOrder` object that is added to the linked list
- For the next test, you may want to use a small regional test file, also turn off any output in the destructor
- Make your code in `main` a comment, the test code goes between your (commented out) code and `return 0;`
- Copy and paste the code for Milestone 4, Test Suite 1 into your function `main()`, run the test

Milestone 5 – April 29th

- Add the modified prototype for the `summaryByType` function and the `summaryByRegion` function
- Copy and paste your implementation code from Project #4 for these functions
- Modify the functions as necessary to use the linked list instead of a vector
- Remove the code for Milestone 4, Test Suite 2
- Make your code in `main` a comment, the test code goes between your (commented out) code and `return 0;`
- Copy and paste the code for Milestone 5, Test Suite 1 into your function `main()`, run the test
  (Note: If this test successfully runs, then you have completed the project. You may submit your program with the test code as if it was your own without attribution.)

Academic Integrity

This is an individual project and all work must be your own. Refer to the guidelines specified in the *Academic Honesty* section of this course syllabus or contact me if you have any questions.

Include the following comments at the start of your source code file:

```
/*
 *  main.cpp
 *
 *  COSC 051 Spring 2019
 *  Project #5
 *
 *  Due on: April 30, 2019
 *  Author: <netID>
 *
 *
 *  In accordance with the class policies and Georgetown's
 *  Honor Code, I certify that, with the exception of the
 *  class resources and those items noted below, I have neither
 *  given nor received any assistance on this project.
 *
 *  References not otherwise commented within the program source code.
 *  Note that you should not mention any help from the TAs, the professor,
 *  or any code taken from the class textbooks.
 */
```

These comments must appear **exactly** as shown above. The only difference will be values that you replace where there are "place holders" within angle brackets such as `<netID>` which should be replaced by your own `netID`. For example, I would replace `<netID>` on the "Author:" line with `waw23`.

Submission Details

Post to Canvas a `.zip` file containing your source code and the given Makefile. Locate the assignment `Project 5` on Canvas and attach/upload your file. Do **not** post your executable file. You should ensure that your source file compiles on the server and that the executable file runs and produces the correct output. Use the following file name for your file: `submit.zip`. The code part of this project is due by end-of-day (11:59pm) on April 30th. Late submissions will be penalized 2.5% for each 15 minutes late. If you are over 10 hours late you may turn in the project to receive feedback but the grade will be zero. In general requests for extensions will not be considered. The value for this project is 100 points.

Programming Skills

The programming skills required to complete this assignment include:

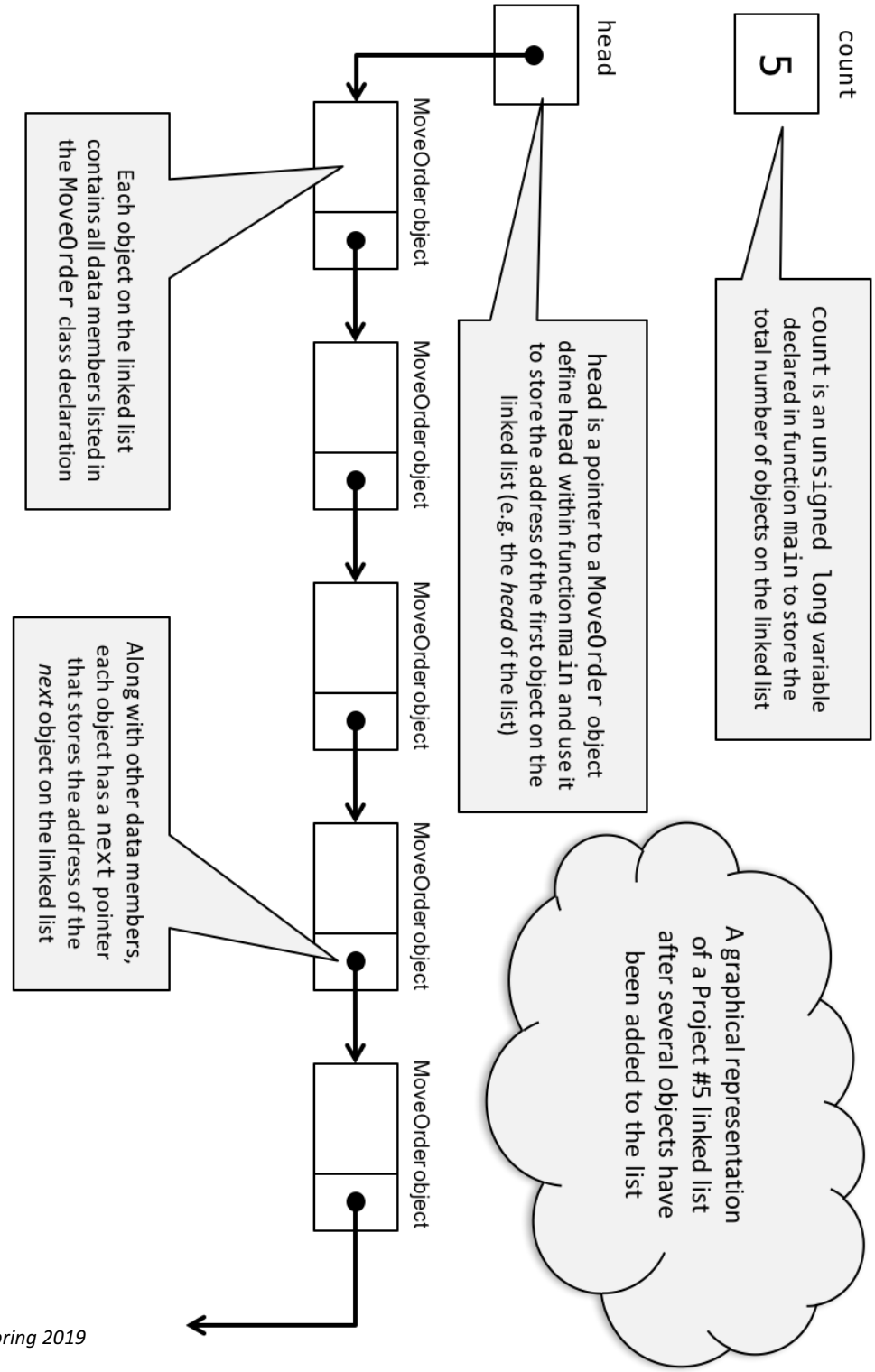
- Input / output (`cin/cout`)
- Basic / advanced data validation
- Basic / advanced output formatting
- Control structures for selection / repetition
- Tabulated output
- Function and calculations
- Object Oriented Design
- Object Oriented Programming
- **Self-referential classes**
- **Pointers and Dynamic memory allocation**
- **Operator overloading**
- **Unit testing / integration testing**

Class `MoveOrder` declaration

Below is a portion of the code from the class declaration and comments about some of the key class members. You should edit your class declaration to reflect the changes / additions shown (or you may download the complete class declaration from Canvas). Refer to the on-line documentation for implementation details for all member functions.

```
class MoveOrder
{
    //overloaded stream insertion operator
    friend ostream& operator<<( ostream &os, const MoveOrder &rhsObj );
    //overloaded stream extraction operator
    friend istream& operator>>( istream &is, MoveOrder &rhsObj );
private:
    //the data members below are required (you may change identifiers)
    //there is no need for additional data members
    int estimateYear;
    int estimateMonth;
    int estimateDay;
    int moveYear;
    int moveMonth;
    int moveDay;
    char type;
    int givenDistance;
    int distance;
    int weight;
    int givenPianos;
    int pianos;
    char stairsOrigin;
    char stairsDestination;
    string estimateNumber;
    string region;
    string customerNameEmail;

    MoveOrder *next;            // New for P5

public:
    //only new or modified member functions are shown below
    //see provided code for additional details

    // constructor with parameters
    // Modified for P5 (may not be in-line)
    MoveOrder(int eYYYY, int eMM, int eDD, int mYYYY, int mMM, int mDD,
              char typ, int dst, int wgt, int pno, char sO, char sD,
              string eNum, string reg, string nameEmail, MoveOrder *ptr = NULL);

    MoveOrder* getNext();              // New for P5 (may be in-line)

    void setNext( MoveOrder *ptr );    // New for P5 (may be in-line)

    ~MoveOrder();    // New for P5 (may not be in-line)

}; //END declaration class MoveOrder
```

Graphical Representation of a linked list of `MoveOrder` objects



(This page intentionally left blank)