Bubba Hotep Moving and Storage, Inc. (BHMSI) – *Object Oriented Version*

Background

Having a menu of options and the capability to load bulk data from files are features of our system that were very well received. Now our corporate executives want to see an object-oriented version. There will not be much difference to actually see (user view), but we are going to modify much of how the software works internally. The menu, file format, data validation, and most of the Project #3 features are unchanged. How these features are implemented is significantly changed.

Software Requirements Overview

We will be adding object-oriented features to the software. Specifically, we shall define a `MoveOrder` class that encapsulates the data and operations applicable to one of BHMSI's move jobs. The complete class declaration is provided on Canvas for download. In-line implementation code is provided for several member functions. You may use this code as your own. Your `MoveOrder` class must have all data members shown in the class declaration. You must also include all of the member functions shown. You may add member functions if they enhance your software. Any such additions must not supersede the intent and/or functionality of required member functions. Data members may not be added. If desired, you may rename data members to match your naming conventions from Project #3. However, you may find it is actually easier not to do so. For class methods, you may use different parameter names, but the number of parameters, their data type(s), and their order in the parameter lists must not be changed. Member function identifiers must not be changed.

The "stand-alone" functions from Project #3 are still required for this project. We shall replace the eight vectors from last project with a single vector. This greatly simplifies the parameter lists for our "stand-alone" functions. The function to display the menu should require no modifications. All of the other functions will need to be changed. Think carefully about the design of your changes before starting to write code. Some modifications may be significant, but with a reasonable amount of prior planning, most changes should be relatively modest or largely copy-and-paste from Project #3.

File Processing (Note changes carefully)

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. Unlike Project #3, all `MoveOrder` objects shall be appended to the vector 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. The complete code for the `printWithMessages()` method is given for you to use without attribution. However, It depends heavily upon your correctly implemented stream extraction operator for the `MoveOrder` class. As you will see in the provided code, the `printWithMessages()` method "knows" how to output data differently for objects with errors and objects without errors. After the detailed data for each row and the summary information have been output the menu shall again be displayed along with a prompt for the user to enter their next option.

Data Validation (no change from previous projects)

Calculations

The actual calculations are unchanged from Project #3. However, where and when these calculations are made is significantly different. In this project, we shall provide `MoveOrder` objects the capability to calculate their own costs for loading, transportation, and extras for pianos and stairs. Depending upon your design for Project #3, you may have had duplicate code to make those calculations in multiple functions (I did). Now we will remove this duplicate code from all of the "stand-alone" functions and encapsulate it within the `MoveOrder` class declaration along with the data necessary for the calculations to be made.

Functions

As before, all code that does "real work" will be in user-defined functions and not part of function `main`. Function prototypes for required functions are shown below. The purpose of each Project #4 function is essentially the same as the corresponding function in Project #3. Beneath each function prototype is an abbreviated description of the function's purpose. Tasks that are unchanged from Project #3 are not repeated. Only changes, additions, and/or deletions from tasks the function must perform are highlighted. As before, functions that require data from the input files should not be called unless at least one file has been successfully loaded.

```
char displayMenu();
```

This function has no parameters and returns a single character. Its purpose is unchanged from Project #3.

```
void loadFile(string fName, bool &loadSuccess, vector<MoveOrder> &vM);
```

The parameters for this function are the input data file name and path, a boolean variable to store true if the file is successfully loaded, and a vector to store one `MoveOrder` object for each line of input data. You may use the identifiers shown, or change them if you prefer other parameter names. File processing is considerably different from Project #3. We shall use the overloaded stream extraction operator of the `MoveOrder` class to extract all values from one row of the file and store those values in a `MoveOrder` object. Data are no longer appended to parallel vectors. `MoveOrder` objects, even those containing data that do not pass validation checks, shall be appended to the end of the single vector parameter of this function. Maintain counters to keep track of how many rows have errors and how many rows are error-free. Output those counts after all rows of data have been processed and displayed. If data from the file are successfully loaded, then the boolean parameter shall be set to true. To decide if the file data were successfully loaded, you can simply test to see if the size of the vector has increased. This is not a very rigorous test, but it is sufficient for our purposes in this project.

```
void allDetails(const vector<MoveOrder> &vM);
```

The parameter for this function is a vector of `MoveOrder` objects. You may change the parameter identifier if you prefer another vector name. The purpose of this function is to display a detailed listing of file data and calculated data, but only for objects that pass all validation checks. The format and content of the output is similar to Project #3. The `validEstimate` method of the `MoveOrder` class shall be called to determine if an object should be displayed. Note that the complete implementation code for the `validEstimate` method is provided for you to use without attribution. Any code that calculated loading or other costs must be removed from the `allDetails` function. Instead, simply call the overloaded stream insertion operator of the `MoveOrder` class. That operator contains the code necessary to generate the correct output, including calculated values.

```
void estimateDetails(const vector<MoveOrder> &vM);
```

The parameter for this function is a vector of `MoveOrder` objects. You may change the parameter identifier if you prefer another vector name. The purpose of this function is unchanged from Project #3. Any code that calculated loading or other costs must be removed from the `estimateDetails` function. Instead, simply call the overloaded stream insertion operator of the MoveOrder class. That operator contains the code necessary to generate the correct output, including calculated values.

```
void summaryByType(const vector<MoveOrder> &vM);
```

The parameter for this function is a vector of `MoveOrder` objects. You may change the parameter identifier if you prefer another vector name. The purpose of this function is unchanged from Project #3. Since the vector contains objects with invalid data, the `validEstimate` method of the `MoveOrder` class must be called to determine if an object is to be included in the summary. Any code that calculates moving or other costs must be removed from the `summaryByType` function. Invoke the appropriate member function(s) of the `MoveOrder` class for any calculated values that are output.

```
void summaryByRegion(const vector<MoveOrder> &vM);
```

The parameter for this function is a vector of `MoveOrder` objects. You may change the parameter identifier if you prefer another vector name. The purpose of this function is unchanged from Project #3. Since the vector contains objects with invalid data, the `validEstimate` method of the `MoveOrder` class must be called to determine if an object is to be included in the summary. Any code that calculates moving or other costs must be removed from the `summaryByRegion` function. Invoke the appropriate member function(s) of the `MoveOrder` class for any calculated values that are output.

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 #4
 *
 *  Due on: April 4, 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 4` 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 4th. 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.

Grade Rubric

The grade rubric for this project will be published separately and posted to the `Project 4` Assignment on Canvas as soon as it is available.

Programming Skills

The programming skills required to complete this assignment include:

- Screen output (`cout`)
- Keyboard input (`cin`)
- Basic data validation
- Basic output formatting
- Basic calculations
- Control structures for repetition
- Advanced output formatting
- Control structures for repetition
- Advanced output formatting
- Tabulated output
- Advanced data validation
- Menu driven programs
- Vectors
- Functions
- **Object Oriented Programming**
- **Operator Overloading**
- **Copy Constructor**

How to approach this program

For this project several milestones are provided. You are NOT required to turn anything in or to meet these milestones. Make sure that your code compiles and runs prior to moving on to the next milestone.

Milestone 1 – NLT March 23

- 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 that you plan to reuse from Project #3

Milestone 2 – NLT March 26

- Copy the provided `MoveOrder` class declaration, and other given code, and paste it into your project
- Write function "stubs" for the class constructors
- Write function "stubs" for any other methods that are not implemented in-line

Milestone 3 – NLT March 26

- Study the project description and bring questions to class
- Copy your loop and the control structure for selection (`switch` or `if / else if`) from function `main` of Project #3 and paste it into your function `main` for this project (this should essentially be everything from your old function main except for declaring the parallel vector objects
- Delete (or comment out) the function calls since the functions are going to change and you will want to implement and test them one at a time, delete (or comment out) references to the old vector identifiers
- Copy and paste the function prototype for `displayMenu` from Project #3
- Copy your function to display the menu from Project #3 and past it into the new project
- Add code to call the display menu function and test to ensure it is still working

Milestone 4 – NLT March 30

- Write your implementation code for the `MoveOrder` class default constructor
- Write your implementation code for the `MoveOrder` class constructor with parameters
- Write your implementation code for the `MoveOrder` class copy constructor
- Write your implementation code for the other `MoveOrder` class member functions
- Write your implementation code for the `MoveOrder` class non-member functions
- Perform unit testing
    - Test A: Instantiate a `MoveOrder` object using the default constructor
    - Test B: Instantiate a `MoveOrder` object using the constructor with parameters
    - Test C: Instantiate a `MoveOrder` object using the copy constructor
    - Test D: Invoke each object's accessor functions to output data member values to the screen and verify that the values are correct
    - Test E: Invoke each object's mutator functions to update the data members and output the new values to the screen to verify that changes were made
    - Test F: Invoke each object's methods that calculate costs; verify that the calculated values are correct
    - Test G: Invoke each object's overloaded stream insertion operator and verify the output is correct
    - Test H: Invoke an object's overloaded stream extraction operator and verify its operation is correct

Milestone 5 – NLT March 31

- Replace parallel vectors in function `main` with one vector of `MoveOrder` objects
- Add the load function to your project and make the appropriate modifications
    - Remove code that used parallel vectors
    - Remove calculations (loading costs, transportation costs, etc.) those should now all be calculated in `MoveOrder` class method `getTotalCost`
    - Replace any stream extraction operations with a single call to the `MoveOrder` object's stream extraction operation to extract each row of file data and update the data members of the `MoveOrder` object, append the object to the vector of `MoveOrder` objects passed in to the reference parameter

Milestone 6 – NLT April 1

- Add the `allDetails` function to your project and modify as appropriate
- Add the `estimateDetails` function to your project and modify as appropriate

Milestone 7 – NLT April 2

- Add the `summaryByType` function to your project and modify as appropriate
- Add the `summaryByRegion` function to your project and modify as appropriate

Milestone 8 – NLT April 3

- Perform integration testing
- Copy to the server and verify that your code compiles and runs in that environment
- Submit your project early!

Class `MoveOrder` declaration

The complete class declaration is provided on Canvas. You may (and should) copy the declaration exactly as it is and use the code as your own without attribution. Below is a portion of the code from the class declaration and comments about some of the key class members.

```
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;
public:  // all member functions for this class are public
         // see provided code, and on-line documentation, for details

}; // END declaration class MoveOrder
```

The class has a default constructor, a constructor with parameters, and a copy constructor. The default constructor should set all data members to default values such as zero for numeric data members, an empty string for string data members, etc. The constructor with parameters shall initialize all data members to the value of the corresponding parameter. The copy constructor shall initialize all data members to the same values stored in the existing `MoveOrder` object passed to its parameter.

Many of the member functions for this class are simple accessor and mutator functions that are implemented in-line. The in-line implementation code is provided as part of the class declaration

(This page intentionally left blank)