CSE 332 Lab 2: Game Boards and Pieces

Due by Thursday October 3rd at 11:59 pm
Final grade percentage: 10 percent

Objective:

This lab is intended to extend your use of basic C++ language features from the previous lab, and to introduce additional features including:

- declaring a struct to represent game pieces (e.g., for games like tic-tac-toe and gomoku),
- using file streams and string streams to import information into a program,
- and subscripting (indexing) into a vector representing the positions of pieces on a game board.

To do this, you will extend your C++ program from the previous lab so that it can
(1) read in and parse files defining the dimensions of a game board, the locations of different pieces on the board, and the symbols used to display the pieces on a game board;

(2) recognize badly formatted inputs that the program can handle and then still proceed (e.g., by skipping over a badly formed line in an input file), but have the program exit gracefully if (and only if) it encounters a case it cannot handle (such as not being able to open the input file at all, or not being able to extract dimensions of the game board from the file); and

(3) if information gathering from the file succeeded, print out a representation of the game board and the pieces on it.

In this lab and in the subsequent labs, you are encouraged to re-use and modify code from your previous lab solution(s) - if you choose to do this, please make a *completely new* copy of any header and source files from your previous lab(s) in the new lab project directory you create. This can avoid some potential for error or confusion within Visual Studio, and also ensures you have a backup copy that you could go back to if a modification you try doesn't pan out. In subsequent lab assignments you will continue to use and extend these features to implement different board games.

Assignment:

Part I - Resources and Guidelines:

1. The C++ string class, file input and output stream, and input and output stringstream examples in the C++ strings, vectors, arrays, and IO lecture slides may be helpful when implementing parts of this lab.

2. The following readings in the textbook may be useful while working on this lab assignment too, and selectively looking up topics and/or reading ahead (on demand as you encounter issues while working on the lab) in the textbook, lecture slides, and studio exercises is also encouraged:
    - C++ program structure and development environment: LLM Chapter 1
    - C++ variables and basic data types: LLM Chapter 2
    - Strings, vectors, arrays: LLM Chapter 3
    - The IO Library: LLM Chapter 8
    - Enumerations: LLM Chapter 19.3

3. **Please re-read the following CSE 332 Programming Guidelines** which are again relevant to this lab as well as the previous one, and please follow them as you implement your solution: **A.1-A.7, A.10-A.17, B.1-B.3, B.8, B.13-B.17, B.20-B.24, B.29, B.31-B.33, and C.1-C.8.**
   In addition, please read the following additional guidelines and please follow them as you complete this assignment: **A.8, A.9, B.10-B.12, B.18, B.19, and B.25-B.28.**

Part II - Program Design and Implementation:

Note: the details of this assignment are again intentionally somewhat under-specified, leaving you some room to choose what you think is the best way to implement them, as long as what you do is reasonable and you explain your design decisions in comments in the code and in your ReadMe.txt file.

Note: this assignment will ask you to use a fairly basic (and perhaps slightly inefficient) approach to containers and the elements they contain, which is based on indexing into a vector using the [] (subscript, also known as index) operator. As we learn more about operator overloading, generic algorithms, associative containers, and other more advanced C++ features, you will be able to use more efficient (and sometimes more straightforward) approaches in subsequent lab assignments.

3. Accept the lab2 assignment from the invitation link here.

4. Open up Visual Studio 2019, connect to GitHub, and clone your newly created repository

to create a local working copy on your h: drive. Create a new Visual C++ Console Application within your repository, make sure to name it something descriptive such as *Lab2*.

5. If you have not done so already in lab 1, create a separate header file and source file for common functions, enumerations, etc. that you are likely to use in any program you write (at least in this course), and move the declaration and definition of your program's usage message function (from lab 1) into those files.

   If you have not done so already in lab 1, please also modify both the declaration and the definition of your usage message function so that it takes in two strings: one giving the name of the program (as you should have done for lab 1 for passing the value of argv[0] from the main function), and another string parameter giving the rest of the information to convey to the user. The usage message function should print out the program name and how to run the program using those parameters, and should return a non-zero integer that is different than the other non-zero return values that are used by the program to indicate other causes of failure. **One helpful trick for ensuring the program doesn't use the same non-zero return value for different causes of failure is to keep those values at a particular place in your program (e.g., as an enumeration as you should have done for lab 1), and each time you identify a new kind of failure add another enumeration label (and use its value at the appropriate place in the code).**

6. In the header and source files you just created to hold common functions, enumerations, etc., also declare and define a function that takes a reference to a C++ style string (e.g., of type string) and modifies that string so that all of the alphabetic characters in it become lowercase (**hints:** variables of type char can be treated as integers that can be added and subtracted, the range of values for 'a' through 'z' is contiguous, and the range of values for 'A' through 'Z' is also contiguous).

7. Create a new header and source file for game pieces, and in the header file declare an enumerated type (for example, named piece_color) for the colors of game pieces. For this program the colors should include red and black and white, though you may also add other colors to the enumeration either now or later in the semester if you'd like. You also should add a label to that enumeration to indicate an invalid color, and another label that indicates no color, both of which may be useful when implementing other parts of this program.

8. In the game pieces header and source files, declare and define a function that takes a single parameter of the enumerated type (e.g., piece_color) by value, and returns (again by value) an **all lowercase** C++ style string corresponding to the value that was passed. For example, passing in a value of piece_color::red should give a return string containing "red".

9. In the game pieces header and source files, declare and define a function that takes a C++ style string by value as its only parameter, converts the passed string parameter (which is a local copy of what was passed by value) and returns (again by value) the element of the enumerated type (e.g., piece_color) that corresponds to that string. The strings "red", "black", and "white" should be recognized as corresponding to the

respective enumeration labels (e.g., piece_color::red, piece_color::black, and piece_color::white respectively). Optionally, you can interpret certain strings (e.g., " " or any other string consisting only of spaces, tabs, or other whitespace) as corresponding to the no color enumeration label, which may be useful for initializing an empty board, etc. in this lab or in future labs. If the string does not correspond to any other color value, the function should return the element corresponding to an invalid color value. For example, passing in a string containing "red" would give a return value of piece_color::red, while passing in a string containing "rook" would give a return value of piece_color::invalid.

10. In the game pieces header and source files, declare and define a struct (named, for example, game_piece) to represent a game piece, which contains a variable of the enumerated type you declared in the step above (e.g., of type piece_color), a (C++ style) string variable to hold the name of the piece, and a (C++ style) string variable to represent how the piece should be displayed when a game board containing it is printed out.

11. Create a new header and source file for a game board, and in them declare and define a function for reading in the dimensions of a game board (row, col) from an input file stream (for example from the first line of either tic-tac-toe.txt or gomoku.txt). The function should return an *int* value and should take three parameters: a *reference to an input file stream* (*ifstream &*) and *references to two unsigned integers*.

    The function should declare a local (C++ style) string variable and then call the C++ IO library's *getline* function with the input file stream and the string. The function should then wrap the string variable in an input string stream and use the input string stream's >> (extraction) operator to extract a value from the string into the first unsigned integer parameter and then use it again to extract another value from the string into the second unsigned integer parameter.

    Be sure to check the value returned by the *getline* function and by each of the extraction operator. If all three of them returned true then the function should return a value indicating success; if the *getline* function returned true but either (or both) of the extraction operators returned false the function should return a unique value indicating it was able to read a line but could not extract the board dimensions; otherwise the function should return a different non-zero value indicating that it could not read a line from the input file stream.

12. In the game board header and source file, declare and define a function for reading in game pieces from an input file stream (for example from subsequent lines of either tic-tac-toe.txt or gomoku.txt). The function should return an *int* value and should take four parameters: a *reference to an input file stream* (*ifstream &*), a *reference to a vector of game pieces* (e.g., *vector<game_piece> &*), and *two unsigned integers* by value.
    The first parameter is for the input file stream from which game piece definitions will be read, the second parameter is for the positions of different game pieces on a game board, and the third and fourth parameters give the horizontal(width or # of rows) and vertical(height or # of columns) dimensions of the game board, respectively.

    The function should declare a local (C++ style) string variable and then repeatedly (until it reaches the end of the input file, which is indicated by the getline function returning false)

call the C++ IO library's getline function with the input file stream and the string. The function should then wrap the string variable in an input string stream and use the input string stream's >> (*extraction*) operator to extract five values from the string: (1) a string for the color of the game piece, (2) a string for the name of the game piece, (3) a string with which to display the game piece when the game board is printed out, (4) an unsigned integer for the horizontal position(x coordinate or column #) of the piece on the game board, and (5) an unsigned integer for the vertical position(y coordinate or row #) of the piece on the game board.

If five values were successfully extracted (the extraction operator will return false if it cannot extract a value), the function should then (1) convert the first string into an enumeration value for the game piece color (by calling the function you wrote earlier to do that) and (2) compare the values of the horizontal and vertical coordinates of the game piece to the horizontal and vertical dimensions of the game board. If the function was not able to extract five values from that line, or if the game piece color is invalid, or if either the horizontal or vertical coordinate is greater than or equal to the corresponding dimension of the game board, then the function should simply skip over that line and continue to process subsequent lines.

Otherwise, the game piece definition on that line is considered to be well formed, and (before moving on to process the next line of the input file) the function should use the horizontal and vertical coordinates that were read from the line (e.g., column # and row #, respectively) and the vertical dimension of the game board (e.g., height) to calculate an index (to use with the [] subscript operator) into the vector of game pieces using a formula that assumes the lower left corner of the game board has coordinates 0,0; and corresponds to the first valid index (index 0) in the vector, and the upper right corner has coordinates width - 1, height - 1 and corresponds to the last valid index in the vector (e.g., index v.size()-1 if v is the name of the vector) according to the formula width * row # + col # (if using cartesian coordinates, row # is the y coordinate and col # is the x coordinate). For the game piece at that index in the vector, the function should then replace its color value with the color value that was read from the line, and similarly should replace its name and the string for printing it out with the corresponding strings that were read from the line.
If the function was able to read even one well formed piece definition line from the input file stream it should return a value to indicate success, and otherwise should return a value to indicate that it failed to find any well formed piece definitions.

13. Declare and define a function for printing out the pieces on a game board, which takes a reference to a const vector of game pieces (for example of type *const vector<game_piece> &*) and two unsigned integers by value as its parameters. This function should print out the layout of the game board as given by the parameters that were passed to it, using the coordinate 0,0 as the lower left hand square of the board (which means that the function will need to print out the vertical dimension in descending order, and the horizontal dimension in ascending order; also, the upper right corner of the board will have coordinate width-1,height-1 where width and height are the horizontal and vertical dimensions of the game board, respectively).
For each coordinate position on the board, the function should print out the string for displaying the piece at the corresponding index in the vector according to the formula

width * row# + col# (as in the previous function). If the function was able to print out the game board it should return a value to indicate success, and otherwise should return a value to indicate the specific reason that it failed (for example the size of the vector might not match the dimensions of the game board that were passed in).

14. Implement your main function for lab 2 as follows:

    - The program should first check that exactly one argument was passed to the program (in addition to the program's name): if not, your program should pass the program name and a second string showing the command line arguments that need to be given, into a call to the usage function described above and then return the (unique non-zero) value that was returned by that function.

    - If the number of command line arguments was correct, use the argument that was passed to the program (after the program name) to construct and open an input file stream. If the input file stream did not open correctly (for example, if a file of the name given does not exist) the program should print out an error message and then return a unique non-zero value corresponding to that kind of failure.

    - Otherwise, declare two unsigned integer variables (one for the horizontal extent(width) and one for the vertical extent(height) of a game board), and call the function for extracting the game board dimensions, with the input file stream and those two variables(width first, followed by height). If (and only if) that function returned a value indicating that it was able to read a line but could not extract the dimensions from that line, the main function should call it again (repeatedly if necessary) until it either returns success or returns a value indicating that it could not read a line from the file - this allows the program to tolerate blank lines or badly formed lines prior to a valid line that contains board dimensions. If the function was never able to extract board dimensions from the file, the program should print out an error message to that effect and return the appropriate non-zero value indicating that kind of failure.

    - Otherwise (if the board dimensions were extracted successfully), the program should declare a vector of game pieces and then for each position on the board (the number of positions is equal to the product of the game board dimensions), repeatedly push back a game piece that represents an empty square on the board (e.g., with no color, an empty name string, and a display string that consists only of a space: " ") into the vector.
      The program should then pass the input file stream, the vector of game pieces, and the horizontal and vertical dimensions of the board into a call to the function that reads game pieces from the input file stream. If that function returns a value indicating failure, the program should print out an appropriate error message and return that value.

    - Otherwise (if at least one game piece definition was successfully read from the file), the program should pass the vector of game pieces and the horizontal and vertical dimensions of the game board into a call to the function that prints out the game

board, and then the program should return the value that was returned by that function, as its own return value.

15. Build your project, and fix any errors or warnings that occur. Please be sure to note all of the different kinds of errors or warnings you ran into (though you don't have to list every instance of each kind) in your ReadMe.txt file. If you were fortunate enough not to run into any errors or warnings, please note that instead, where you would have listed errors or warnings in your ReadMe.txt file.

16. Open up a Windows console (terminal) window and change to the directory where your program's executable file was created by Visual Studio.

17. Run the executable program through a series of trials that test it with good coverage of cases involving both well formed and badly formed inputs.
    For this lab one especially useful test is to check whether the output printed by your program consistently represents the contents of the input file it was given. Also, you should add and test examples of badly formatted lines, blank lines, etc. in the input file to make sure your program can handle them and keep going (and correctly process all well formed lines that appear later in the file).
    In your ReadMe.txt file please document which cases you ran (i.e., what the command lines were and what additional cases you added to the files) and summarize what your program did and whether that is correct behavior, in each case.
    You should first make sure the program behaves correctly with the provided files, as in:

    lab2.exe tic-tac-toe.txt

    or
    lab2.exe gomoku.txt

    And then make new versions of those files with badly formed lines and blank lines to test your program's handling of those inputs as well.

18. In your ReadMe.txt file, make sure that your name and the lab number (lab 2) are at the top of the file, and that you've documented whether you ran into any errors or warnings while developing your solution (and if so what they were) and what the executable program did for each of the trials you ran. Be sure to save your changes to your ReadMe.txt file, as well as those to your source (and header) file(s) before committing your final solution.

19. Stage and Commit your changes with a useful commit message. Once committed, push your commit to your remote repository on GitHub. Log on to GitHub and ensure your code was pushed successfully. **If you forget to push your code, it cannot be graded!**

Part III - Extra Credit: (optional, worth up to 5% extra credit)

The goal of this optional part of the assignment is to make further use of the structs, containers, and functions you have defined, to print out how many of each different kind of neighbor each

piece has.

For extra credit, please do the following:

20. Declare and define a function that takes the same list of parameters as the function that prints out the game board, and on a separate line for each piece on the game board prints out the square, name, and color of that piece, followed by a colon, followed by a semicolon-separated list of pieces occupying squares adjacent to it (vertically, horizontally, or diagonally). For example, if there were a black tic-tac-toe stone at 0,0 and 0,2 and a white stone at 1,1, a reasonable output of this function could be:

    0,0 black stone: 1,1 white stone
    1,1 white stone: 0,0 black stone; 0,2 black stone
    0,2 black stone: 1,1 white stone

21. After calling the function that prints out the game board, your main function should call this new function with the appropriate parameters.

22. Test your implementation with different arrangements of pieces on a board including cases involving squares in the corners, on the edges, and in the middle of the board.

23. add a section marked "Extra Credit" at the end of the ReadMe.txt file you submit with your solution, and in that section summarize how you designed, implemented, and tested the extra credit features. Please ask for help from your instructors or teaching assistants if you are uncertain about how to do this extra credit part or if you run into any difficulty with it.

Project Designed by Chris Gill