CSE 332 Lab 4: Multiple Games

Due by Tuesday November 12th at 11:59 pm
Final grade percentage: 15 percent

Objective:

This lab is intended to extend your use of basic C++ language features from the previous labs, and to give you more experience using object-oriented programming techniques in C++, including:

- refactoring common data structures and behaviors into a base class, and
- specializing how the common parts are used in different derived classes.

To implement this lab, you will extend your C++ program from the previous lab so that, in addition to playing the TicTacToe Game from lab3, it also can:

(1) display and play Gomoku game in a larger board to get 5 stones in a row; and
(2) detects successful completion of the game or a draw.

In this lab, you will need to refactor your lab 3 implementation so that the code and data structures from both the TicTacToe game (which you implemented in the previous lab) and a new Gomoku game (which you will implement in this lab), form a cohesive class inheritance hierarchy, supporting a non-trivial degree of re-use of the data structures and code that are common to both games.

For this lab, **you may work individually or in a 2-person or 3-person group.** If you choose to work in a group, you work within a single repository, but in the ReadMe.txt file you then must:

(1) indicate who the members of the group are, and
(2) explain in some detail how you divided up the work (reasonably) evenly between you in designing and implementing your lab solution - i.e., who did which parts and how each of those parts contributed to the submitted solution.

Note that if you do work in a group you are free to use any one of your lab 3 solutions or any combination of them as the starting point for lab 4.

As in the previous labs, when you move code from a previous lab solution's project, please put a completely new copy of any header and source files from your previous lab(s) into the new lab project directory you create, and then add those files to the project (using the Add->Existing menu item). This can avoid 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.

Assignment:

Part I - Resources and Guidelines:

1. The constructor, destructor, operator, accessor, and mutator method examples in the C++ classes lecture slides give an overview of the kind of class syntax and structure you may need for this lab.

2. The examples on C++ subclassing and inheritance polymorphism in these slides also may be useful for thinking about how to structure your base and derived classes, and you also are encouraged to read ahead in the course text book about those topics, particularly in LLM Chapter 15 (up through section 15.6). Reading ahead through LLM Chapter 12.1 (dynamic memory allocation and smart pointers) also is strongly encouraged.

3. Please read the following CSE 332 Programming Guidelines which are relevant to the previous labs and please follow them as you implement your solution: **A.1-A.17, B.1-B.3, B.10-B.17, B.20-B.24, B.29, B.31-B.33, and C.1-C.8.**
   Please also review and follow these additional guidelines that may be relevant to this lab: **B.4-B.5** (dynamic memory allocation), **B.9** (virtual destructors), **B.25** (circular dependencies), **B.28** (narrow interfaces), and **B.34** (smart pointers).

Part II - Code Refactoring:

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.

4. Before accepting the lab 4 assignment, decide who your group members will be and decide on a team name. **Send an email directly to the instructor (shidalj@wustl.edu) with the subject line "CSE332 Lab 4 Group" that includes your team name and each group member's name**. One of your group members should now accept the assignment **here** and create a new team with the decided upon name. Once the repository is created, the other team members should go ahead and accept the assignment. Make sure to join the existing team so you all have shared access to the same repository.

5. Open up Visual Studio, and create a new Console Application C++ project within your repository for lab 4. Copy any source and/or header files you want to use from the previous lab into your new repository under the appropriate folder under that project's top level folder, and then within Visual Studio add them to the project.

6. *Refactor* your game class from the previous lab (e.g., of type TicTacToeGame) into a base class (e.g., of type GameBase) that contains **protected** member variables that are common to multiple games (i.e., the game board dimensions, the containers, etc.), and a class derived from that base class through public inheritance (e.g., of type

TicTacToeGame) that implements the TicTacToe game. For now only move member variables up to the base class, though in subsequent steps you will also refactor some of the methods between the base and derived classes.

7. If you have not already done so, modify the handling of symbols for displaying game pieces so that a (potentially multi-character) C++ style string is used instead of a single char, to display each piece when the board is printed.

8. Modify your base class so that it remembers the longest display string length of any piece currently on the board, in a protected member variable. Make sure that member variable is updated if needed whenever a piece is added to the board. This will be used to ensure columns of our board are aligned correctly when printed.

9. Modify the ostream insertion operator (from the previous lab) so that it takes a reference to a const object of the derived class for your TicTacToe game, and uses the maximum display string length stored by the base class to print the board out not just by putting a space between consecutive piece symbols on a line, but also adjusting the widths of the squares according to the maximum display string length so that columns of the board are aligned vertically with at least a space between pieces when they are displayed. **Hint:** the setw manipulator (and possibly other stream manipulators) and/or the stream width method can help to simplify coding of that feature.

10. Declare a *pure virtual* print() method in the base class that takes no parameters and has a void return type (i.e., virtual void print() = 0;)

11. In the derived class for your TicTacToe game, **override** that virtual print() method and define it as simply inserting *this into an ostream (so that it uses the insertion operator described above).

12. Make the done() method of your derived class virtual, and in the base class declare the same method signature to be *pure virtual*.

13. Make the draw() method of your derived class virtual, and in the base class declare the same method signature to be *pure virtual*.

14. Move the prompt() method to the base class.

15. Make the turn() method of your derived class virtual, and in the base class declare the same method signature to be *pure virtual*.

16. Move the play() method from your derived class to the base class and modify it so that instead of printing the game board using the ostream operator directly, it instead calls the print() method (that allows each derived class to override printing using its own ostream insertion operator).

17. Add a static method to the base class that takes an integer and an array of pointers to char (the same types as are given to argc and argv in your program's main function signature) and returns a pointer to an object of your base class. The method should check

that exactly one argument has been passed to the program (in addition to the program's name), and if so that the string in argv[1] is "TicTacToe". If so, the method should use the new operator to dynamically default construct an object of your derived TicTacToe game class, and return the address of that object; otherwise, the method should return a singular pointer (i.e., a pointer whose value is 0), indicating that no object was constructed.

18. Modify your program's main function so that it passes argc and argv into the static method described above, and if the pointer returned from that function call is singular (has a value of 0) returns the result of calling the usage function from the previous lab; otherwise if the pointer was non-zero the main function should use the pointer to invoke the allocated object's play() method and use the result returned by the play() method as the program's return value.

    Before exiting, the main function must make sure that if a game object was allocated (i.e., the pointer that was returned by the static method was non-zero) the allocated object is also destroyed after the call to the play() method has completed. One good way to do that is to initialize a shared_ptr variable with the pointer that was returned by the static method: when the main function exits the shared_ptr is destroyed and its destructor will call delete on the pointer with which it was initialized, which will destroy the dynamically allocated object, safely and reliably.

    As before, the main function also should catch any exceptions thrown by the object's methods, and return an appropriate unique non-zero value from the program for each case. With dynamic memory allocation, make sure to catch the bad_alloc exception that might be thrown by the new operator if it fails to allocate memory.

    As in the previous lab, in this lab your main program should only return 0 if the game being played was successfully completed, and otherwise (e.g., if the user quit or the game reached a point where successful completion was no longer possible) should return a unique non-zero value for each different failure case.

19. **Before proceeding to the rest of the assignment, please make sure your refactored TicTacToe game performs correctly**, and before submitting your lab 4 solution, please also make sure that you fix the issues that were raised in the lab 3 grading comments that were returned to you.

Part III - Adding a Second Game:

20. Declare and define a Gomoku game class (e.g., of type GomokuGame) that is derived through public inheritance from game base class (e.g., of type GameBase).
    The game board should be a 19 by 19 square (for extra credit you can generalize this to an n by n square). Each piece of the game is either a black stone or white stone and should be displayed as "B" or "W". Black stone will always go first.

21. Update the static method of the base class that parses argc and argv so that if the string in argv[1] is "Gomoku" it should dynamically default construct an object of your derived Gomoku game class, and return the address of that object.

22. Outside your Gomoku game class, declare and define an insertion operator (operator<<) that takes a reference to an ostream and a reference to a const Gomoku class object as parameters, and returns the ostream reference that was passed in (that allows the operator to be called repeatedly on the same ostream object as in cout << gomoku_game << endl;). Inside your Gomoku game class, declare the insertion operator to be a friend so that it can access the private member variables of the game object. The operator should print out the current state of the game board with each piece's display string for squares that have numbered pieces on them, and spaces for empty squares, with labels on the axes. For example, initially the board would be printed out as:

    19
    18
    17
    16
    15
    14
    13
    12
    11
    10
    9
    8
    7
    6
    5
    4
    3
    2
    1
    X 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

23. In the derived class for your Gomoku game, override the base class print() method and define it by simply inserting `*this` into an ostream (so that it uses the insertion operator for that class, which is described above).

24. Declare and define a public virtual (non-static) done() method that takes no parameters (other than the this pointer that's passed implicitly to all non-static methods and operators) and has a bool return type. The method should return true if 5 stones of the same color are in a row or column or diagonal (otherwise the method should return false.

25. Declare and define a public virtual (non-static) draw() method that takes no parameters(other than the this pointer that's passed implicitly to all non-static methods and operators) and has a bool return type. The method should return true if there is no path left that can lead to 5 stones in a row, otherwise the method should return false.

26. Provide a public virtual (non-static) prompt() method that behaves exactly as the corresponding prompt method in your TicTacToe game class. One easy way to implement this is to declare the prompt method virtual instead of pure virtual in the base class, and move the implementation of the prompt method from the TicTacToe game class up to the base class so that it is available to the Gomoku game class as well.

27. Declare and define a virtual public (non-static) turn() method that takes no parameters and has an appropriate return type. The method should call the overloaded prompt() method to obtain coordinates for an empty square on the board and place the appropriate piece (alternating "Black Stone" and "White Stone") on that square, and return.

Part IV - Testing and Submitting Your Solution:

28. 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.**

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

30. 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 on the command line and from the user.

31. In your ReadMe.txt file please document which cases you ran (i.e., what the command lines were, what you tried while playing the game, etc.) and summarize what your program did and whether that is correct behavior, in each case.
    You should first make sure the program behaves correctly when run with a well-formed command line, as in:

    lab4.exe TicTacToe

    or

    lab4.exe Gomoku

    Make sure your program can handle valid and invalid move inputs from the user and keep on playing, that the program allows the user to quit when they choose, and that the program identifies successful completion (indicated by the done() method) or unsuccessful completion (indicated by the draw() method) of the game.
    Then test your program's handling of badly formed command lines (e.g., without a game name, or with a game name other than "TicTacToe" or "Gomoku").

32. In your ReadMe.txt file, make sure that all group members names and the lab number (lab 4) 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 when you tested your program. **If your repository contains files (.h or .cpp files that you wrote) that are not used in your final solution,** list in your readme.txt file all files that are used in your solution.

33. Commit and push your completed lab to your group's GitHub repository.

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

The goal of this optional part of the assignment is to generalize your Gomoku game so that it can be played with an arbitrary sized board (greater than or equal to 3 by 3) and with an arbitrary number of connecting pieces in a row to win(>= 3). For extra credit, please do the following:

34. Modify your Gomoku class (updating constructors, etc. as needed) so that it can be played with an arbitrary sized square game board (greater than or equal to 3 by 3).

35. Modify your Gomoku class (updating constructors, etc. as needed) so that it can be played with an arbitrary number of connecting pieces needed to win (the arbitrary number is greater than or equal to 3).

36. Modify the static base class method that parses argc and argv so that for the Gomoku game one or two additional arguments can be specified after the game name. If neither of them is specified, the game should play as before with a 19 by 19 game board and 5 as the number of pieces in a row is needed to win.
    If one additional parameter is specified, it should be interpreted as the number of squares on a side of the board: for example if 5 were given, the board would be 5 by 5
    If two additional arguments are specified, the first should be interpreted as the number of squares on a side of the board and the second as the number of connecting pieces in a row/column/diagonal needed to win: for example if 10 and 4 were given, the board would be 10 by 10 and 4 pieces in a row would be needed to win.

37. Test your implementation to make sure that with no additional arguments after the game name, with a single extra argument of 3, or with two extra arguments of 3 and 3, that the Gomoku game still plays as before the extra credit portion was added. Also test different combinations of arguments beyond those to make sure your game plays those cases correctly as well. Also make sure that if zero or a negative number is given for the board size parameter that the program notices it, issues an error message, and returns a unique non-zero value for that case.

38. 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. In that section also show the output produced by your program for one run with two additional arguments that specify both a different board size than 3 by 3

and a different connecting pieces value than 3.
Please ask for help from your instructors or teaching assistants if you are uncertain about how to do the extra credit part or if you run into any difficulty with it.

Project Designed by Chris Gill