CSE 20212 Fundamentals of Computing II
Spring 2016

Lab #1 handout (Week of January 18)
Due 3pm before your next lab, **no exceptions**!

Required reading:

Week of 1/18: Chapters 9 and 10 in D&D
Next week: Chapter 10 and Chapter 11 in D&D

Objectives

1. Get more C++ experience via a simple game
2. Use composition while developing simple classes
3. Try the new/ delete operators in the context of a constructor/deconstructor
4. Have fun!

Board games (revisited)

NOTE: Given the heterogeneity of programming experience and new transfers, this first lab is more regimented than you may be used to. Much of the specificity is an attempt to limit time outside of lab while reinforcing the objective of this lab: an intro to composition. If you would like to express more creativity, we have added an extra credit option in Part 3. If you still want to change things up, please refer to the posted grading rubric and ask us questions as needed.

Part 1: Column class (should be completed in-lab)

1. Report to lab **on time**. Attendance will be taken at the scheduled lab time.

2. Read the instructions for labs below, and ask your TAs or instructor if you have any questions about preparing or submitting your lab:

   https://www3.nd.edu/courses//cse/cse20212.01/www/labs.html

3. Connect Four (http://en.wikipedia.org/wiki/Connect_Four) is a simple game where two players take turns placing colored discs into a seven-column, six-row grid. Today, we will implement a simple version of Connect Four using composition.

4. First, we will develop a class called C4Col, which will be responsible for simply storing discs placed into columns. C4Col should contain three private data members: an integer storing the number of discs placed in the column (so far), an integer storing the maximum number of discs allowed (i.e., # of rows = 6), and a simple character array to store representations of each players' discs (more later). Remember: initialization must be done in the constructor, not the .h file

5. Develop an interface (i.e, prototypes of the member functions for this class in your

.h file) composed of a default constructor and the following member functions: int **isFull**(), which determines if the column is full (i.e., numDiscs == maxDiscs); char **getDisc**(int), which returns the requested element of the private character array (i.e., getDisc(0) will return Discs[0]); int **getMaxDiscs**(), which returns the maximum number of discs (i.e., number of rows); and void **addDisc**(char), which adds the character representing a disc to the next open slot in the Disc array (i.e., Discs[numDiscs++] = newDisc). Remember: developing an interface simply means adding function prototypes inside a new class .h file as we have done in lecture. Although there are hints on how to implement these, the functions go in a separate .cpp file (see 5 and 6 below).

6. Implement a default constructor for the C4Col class in your implementation file (.cpp). This default constructor will initialize the current and maximum number of discs to 0 and 6, respectively, and initialize the character array with ' ' characters.
   REMINDER 1: constructors have no type; C4Col::C4Col(){} is an empty constructor. All member functions should have a type, e.g., int **getDisc**(int);
   REMINDER 2: discs[i] = " "; is a compiler error… it must be ' ' in C++.

7. Implement the member functions listed (and hinted at on page 1) in your emerging .cpp file. If **addDisc** is called and the column is full, display a message but do nothing else. If the parameter given to **getDisc** is invalid, also display an error message to the user and do nothing.
   (hint: you should use **isFull**() in **addDisc**() for maximum code reusability)

Part 2: Board class

1. The second class will represent a Connect 4 board and contain data members that are C4Col objects. This concept is called composition. You are encouraged to work on your Connect4 game if you can in lab.

2. Develop the C4Board class starting with an interface (.h) that includes two private data members: an integer for the number of columns and an array of C4Col objects to represent the Connect 4 board (e.g. C4Col Board[100]; the constructor you developed in part 1 will be automatically run for all elements of this array). Next, add prototypes in your interface for a default constructor and two public member functions: void **display**(), which will display the current board in simple text; and void **play**(), which will allow two players to play a game. REMINDER: you will need to also include the header developed in part 1 here to declare C4Cols like this:

   #include "C4Col.h"

3. Implement the member functions for this C4Board class. The default constructor should set the number of columns to be the row size of a Connect 4 board (n = 7). As mentioned above, constructors for the composed class (C4Col) are run automatically. **Display**() could contain a nested for loop that decreases from numRows – 1 to 0 in the outside loop and from 0 to numCols – 1 on the inside as:

```
for (int i = board[0].getMaxDiscs() - 1; i >= 0; i--) {
   …
   for (int j = 0; j < numCols; j++)
     cout << board[j].getDisc(i) << " ";
   …
}
```

NOTE: You must include "C4Col.h" and "C4Board.h" in this new .cpp file

Add extra formatting to the above to make the board look proper including a separator character ('|') between columns, a number indicating which column is which for playing the game, and other enhancement (use your creativity here). **Play**() should display the current board (using display to maximize code reuse) and ask one of two players which column they would like to add their disc, or -1 to end the game. Use an 'X' character for player 1 discs and an 'O' character for player 2 discs (HINT: (turn % 2)+ 1 will give you the appropriate player if player 1 always goes first). Use **addDisc** from the C4Col class to complete the turn.

4. Your main function/client program should be very simple. Declare a C4Board object (as we need at least one instantiated to use member functions), and then call its **play** member function at a minimum like this:

```
/*  insert comments here */
#include "C4Board.h"   // class definition for C4Board used below

int main() {
  C4Board c4;   // instantiate an instance of a C4Board object
  c4.play();       // play game!!
}
```

Part 3: Finishing up

- Modify your C4Col and C4Board classes to use dynamic memory allocation. Change the appropriate variables to be appropriate pointers (i.e, char * for C4Col) and use the operation "new" to allocate the appropriate number of elements. Add a deconstructor that calls "delete []" whenever these objects are destroyed. Please read the text (or online) about new/delete and refer to lecture notes if you need to.
- Finish your Connect4 game by adding a private member function (called a helper function in the text) that can determine if a player has won and, if they have, displays a congratulatory message. If you have trouble, please see the instructor or the TAs. HINT: This is similar to the 2-D array manipulation you did for Lab 6 in the Fall, but A[i][j] becomes A[i].getDisc(j) using composition.
- Kick it up a notch (10% extra credit): Develop a computer player that follows your move with a move of its own. Feel free to make this simple (choose a random non-full column) or as sophisticated of an AI as you'd like.

Also, in your report discuss in your own words, what are the advantages of using a "get" function for a data member like maxDiscs in your implementation of Connect4? Why are deconstructors needed with dynamic memory management?

Grading rubric for Lab 1:

Please note: if your lab program(s) does not compile because of unfixable problems, AT MOST 50 percent of the available grade will be awarded based on how complete the lab assignment is. If you are having trouble, please come to our day or evening office hours earlier rather than later and we'd be happy to discuss the issues with you.
Part 1: 25 points

+2 Makefile correctly compiles code
+6 Basic program structure is correct
- program commented well (3 pts)
- proper .h and .cpp files (3 pts)
+3 default constructor works as requested
+10 member functions are correct (2 pt each)
- isFull()
- getDisc()
- getMaxDiscs()
- addDisc()
+4 Invalid inputs checked in addDisc and getDisc (2 pts each).

Part 2: 30 points

+8 Basic program structure is correct (2 pts each)
- commented well
- proper variable names
- includes proper header files
- interface separated from implementation
+2 default constructor works as intended
+10 Board is displayed correctly.
+10 plays game correctly with 'X' and 'O characters for player 1 and 2. respectively.

Part 3: Finishing up 30/pts

+4 uses new properly in the constructor
+4 uses delete properly in the deconstructor
+8 Checks for legal/illegal moves correctly.
- you will only be expected to check invalid integer input; we will not put chars or strings in the game for testing
+14 Correctly determines when game is over.

Extra credit: Any computer player, no matter how simple, will receive 10% extra points relative to Parts 1 - 3. For example, a score of 80/85 would receive 8 extra credit points, making the score 88/85.

Lab Report: 15 points
+3 Explains how the user uses the program.
+6 Explains how the program works internally including one of the following:
one of the below will be graded for 2 points
- if extra credit is done, explains your implemented approach for extra credit
- if extra credit was not done, describes how your determination of a winner works
+3 Explains how the program was verified.
+1 Explains the advantages of using a get function?
+2 Explains why deconstructors are needed with dynamic memory management

Common compiler issues to keep in mind

- Static arrays must have a number of elements defined, even if it is more than you need. char discs = char[]; will compile but likely seg fault. Use something like this to guard against issues: char discs = char[100];
- When you debug your code, start with the first listed issue from the compiler. Something as simple as a missing ";" may generate tens of errors in your code.
- Delete by default does not know an object is an array. You need to instruct it to remove multiple entries by using [] in your deconstructors.
- Variables in C++ are case-sensitive. Board and board are different in your code.
- Whenever you design an interface (.h), you should preface it with "#ifndef <HEADER>" and end it with "#endif." This ensures that the class definition is included only one time during compilation even if you #include it multiple times.
- Member functions require () after them even if there are no arguments. **isFull** will not compile while **isFull**() will.
- For now, add "using namespace std;" after your includes, as we will only use the standard namespace throughout the semester.

Makefile you must include (file names can differ):

* Please see the lecture notes for 1/15 as copy and paste from a PDF has caused problems in the past for this problem/course. Let us know if you have issues.

Coder challenge! (board game edition)

An interesting problem for chess buffs is the *n* Queens problem: is it possible to place *n* queens on a chessboard such that no queen is threatening another queen? Practically, this means no two queens are in the same row, column or along the same diagonal. Note that there are over 4.4 billion possibilities for placing eight queens (64 choose 8) on a regular chess board but only 92 valid solutions.

Develop a C++ program to find at least one solution. Solving and outputting at least one solution for the Eight Queen problem (*n*=8) likely requires recursive solutions and will be awarded up to $40 coder dollars determined by our course software manager.