Assignment 4: Boggle

Thanks to Julie Zelenski and Eric Roberts for creating this assignment, and to Chris Piech and Marty Stepp for modifications.

JULY 19, 2017

The classic CS106B assignment is based off a real life game called Boggle. Image courtesy of Rich Brooks (Flickr)

It's time for a CS106 classic, the venerable word game Boggle! The Boggle game board is a square grid onto which you randomly distribute a set of letter cubes. The goal is to find words on the board by tracing a path through adjoining letters.

Your assignment is to write a program that plays a fun, graphical rendition of this little charmer, adapted for the human and computer to play pitted against one another. You'll quickly learn you have little hope of defeating the computer, but it's awesome to realize you wrote this program that can so soundly thrash you again and again!

This assignment will introduce you to classes, but the main focus is on designing and implementing recursive algorithms. The starter code for this project is available as a ZIP archive. A demo is available as a JAR.

Starter Code

Demo Jar

Turn in the following files:

1. **boggle.cpp**, the C++ code for all of your recursion functions.
2. **boggle.h**, the header file with your function definitions and class variables.
3. **boggleplay.cpp**, the play interface for the game.

This is a pair assignment, so you may work in a pair or alone. Find a partner in section or use the course message board. If you work as a pair, comment both members' names atop every code file. Only one of you should submit the program; do not turn in two copies. Submit using the Paperless system linked on the class web site.

Due Date: Boggle is due Wednesday July 26th at 12:00pm.
Y.E.A.H hours: Video: TBA
Related Reading: The Boggle Game; C++ Classes

Game Overview

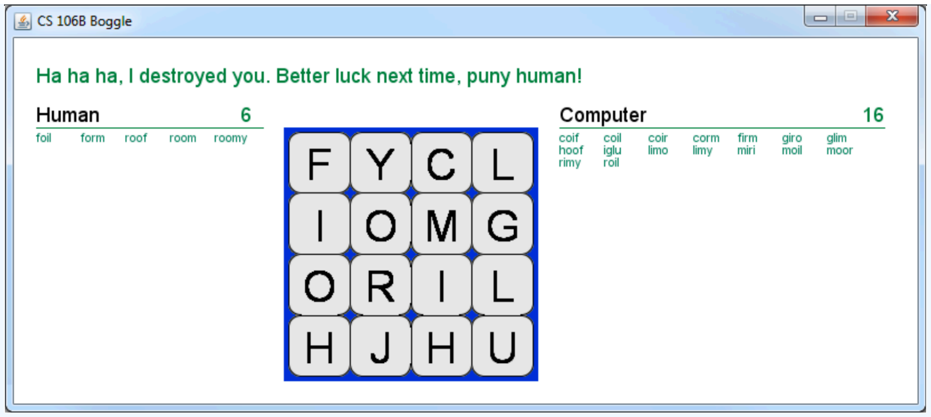
Boggle is a game played on a square grid onto which you randomly distribute a set of letter cubes. Letter cubes are 6-sided dice, except that they have a letter on each side rather than a number. The goal is to find words on the board by tracing a path through neighboring letters. Two letters are neighbors if they adjoin each other horizontally, vertically, or diagonally. There are up to eight letters neighboring a given cube, and each cube can be used at most once in a word. In the real-life version of this game, all players work at the same time, listing the words they find on a piece of paper. When time is called, duplicates are removed from the lists and the players receive one point for each unique word, that is, for each word that player found that no other player was able to find.

You will write a program that plays this game, adapted for one human to play against a computer opponent. Unfortunately, the computer knows recursive backtracking, so it can find every word on the board and destroy you every time. But it's still fun to write a program that can resoundingly beat you again and again.

To begin a game, you shake up the letter cubes and lay them out on the board. The human player plays first, entering words one by one. Your code first verifies that the word is valid, then you add it to the player's word list and award the player points according to the word's length (one point per letter ≥ 4). A word is valid if it meets all of the following conditions:

1. at least 4 letters long
2. is a word found in the English dictionary
3. can be formed by connecting neighboring letter cubes (using any given cube only once)
4. has not already been formed by the player in this game (even if there are multiple paths on the board to form the same word, the word is counted at most once)

Here is an excerpted sample log of execution for a round of the game:



```
CS 106B Boggle

Ha ha ha, I destroyed you. Better luck next time, puny human!

Human                         6
foil   form   roof   room   roomy

F Y C L
I O M G
O R I L
H J H U

Computer                                        16
coif   coil   coir   corm   firm   giro   glim
hoof   iglu   limo   limy   miri   moil   moor
rimy   roil
```

```
Do you want to generate a random board? y
It's your turn!
FYCL
IOMG
ORIL
HJHU

...

Your words (3): {"FOIL", "FORM", "ROOF"}
Your score: 3
Type a word (or Enter to stop): room
You found a new word! "ROOM"

...

Your words (5): {"FOIL", "FORM", "ROOF",  "ROOM", "ROOMY"}
Your score: 6
Type a word (or Enter to stop):
It's my turn!
My words (16): {"COIF", "COIL", "COIR", "CORM", "FIRM", "GIRO", "GLIM"
"HOOF", "IGLU", "LIMO", "LIMY", "MIRI", "MOIL", "MOOR", "RIMY", "ROIL"}
My score: 16
Ha ha ha, I destroyed you. Better luck next time, puny human!
```

As the log shows, once the player has found as many words as she can, the computer takes a turn. The computer searches through the board to find all the remaining words and awards itself points for those words. The computer typically beats the player, since you will program it to find all possible words left.

Your program's output format should exactly match the abridged log of execution above. Here are more examples of game logs:

Boggle Run #1

Boggle Run #2

Boggle GUI #1

Boggle GUI #2

Boggle GUI #3

Setting up the Game Board

The real Boggle game comes with sixteen letter cubes, each with particular letters on each of their six faces. The letters on each cube are not random; they were chosen in such a way that common letters come up more often and it is easier to get a good mix of vowels and consonants. We want your Boggle game to match this. The following table shows the letters represented on each of the sixteen cubes from the original Boggle (i.e., all six faces for each of the cubes). You should decide on an appropriate way to represent this information in your program and declare it accordingly.

AAEEGN	**ABBJOO**	**ACHOPS**	**AFFKPS**
AOOTTW	**CIMOTU**	**DEILRX**	**DELRVY**
DISTTY	**EEGHNW**	**EEINSU**	**EHRTVW**
EIOSST	**ELRTTY**	**HIMNQU**	**HLNNRZ**

At the beginning of each game, "shake" the board cubes. There are two different random aspects to consider:

1. A random location on the 4x4 game board should be chosen for each cube. (For example, the **AAEEGN** cube should not always appear in the top-left square of the board; it should randomly appear in one of the 16 available squares with equal probability.)
2. A random side from each cube should be chosen to be the face-up letter of that cube. (For example, the **AAEEGN** cube should not always show **A**; it should randomly show **A** 1/3 of the time, **E** 1/3 of the time, **G** 1/6 of the time, and **N** 1/6 of the time.)

The Stanford C++ libraries have a file shuffle.h with a shuffle function you can use to rearrange the elements of an array, Vector, or Grid. See shuffle.h if you are curious about how the shuffling algorithm works.

Your game must also have an option where the user can enter a manual board configuration. For this option, rather than randomly choosing the letters that will be on the board, the user enters a string of 16 characters, representing the cubes from left to right, top to bottom. (This is also a useful feature for testing your code.) Verify that the user's string is long enough to fill the board and re-prompt if it is not exactly 16 characters in length. Also re-prompt the user if any of the 16 characters is not a letter from A-Z. Your code should work case-insensitively. You do not need to check whether the 16 letters typed could actually be formed from the 16 letter cubes; just accept any 16 alphabetic letters.

Human Player's Turn

The human player enters each word she finds on the board. As we described above, for each word the user types, you must check that it is at least four letters long, contained in the English dictionary, has not already been included in the player's word list, and can be formed on the board from neighboring cubes. If any condition fails, alert the user. There is no penalty for trying an invalid word, but neither do invalid words count toward the player's list or score.

If the word is valid, you add the word to the player's word list and score. The length of the word determines the score, with each letter ≥ 4 being worth 1 point. For example, a 4-letter word is worth 1 point; a 5-letter word is worth 2 points; 6-letter words are worth 3; 7-letter words are worth 4; and so on. The player enters a blank line when she's done finding words, which signals the end of the human's turn.

Computer's Turn

Once the human player is done entering words, the computer then searches the entire board to find the remaining words (that is, the words missed by the human player). The computer earns points for each remaining word found that meets the requirements (minimum length, contained in English lexicon, not already found, and can be formed on board). If the computer's resulting score is strictly greater than the human's, the computer wins. If the players tie or if the human's score exceeds the computer's, the human player wins.

You can find all words on the board using recursive backtracking. The idea is to start from a given letter cube, then explore neighboring cubes around it and try all partial strings that can be made, then try each neighbor's neighbor, and so on. The algorithm roughly looks like the following:

```
for each letter cube c:
    mark cube c as visited. // choose
    for each neighboring cube next to c:
        explore all words that could start with c's letter. // explore
    un-mark cube c as visited. // un-choose
```

Implementation Details

You will write the following two sets of files:

`boggleplay.cpp`: client to perform console UI and work with your Boggle class to play a game

`Boggle.h/Boggle.cpp`: files for a Boggle class representing the state of the current Boggle game

In this section we describe the expected contents of each of these in detail.

---

boggleplay.cpp

We have provided you with a file **`bogglemain.cpp`** that contains the program's overall main function. The provided code prints an introduction message about the game and then starts a loop that repeatedly calls a function called **`playOneGame`**. After each call to **`playOneGame`**, the main code prompts to play again and then exits when the user finally says "no". The **`playOneGame`** function is not provided; you must write it in **`boggleplay.cpp`**. In that same file, you can place any other logic and helper functions needed to play one game. You may want to use the **`getYesOrNo`** function from simpio.h, which prompts the user to type yes/no and returns a **`bool`**.

One aspect of the console UI is that it should "clear" the console between each word the user types, and then re-print the game state (including the board, words found so far, score, etc.). This makes a more pleasant UI in which the game state is generally visible at the same place on the screen at all times during the game. See the provided sample solution for an example. Use the Stanford Library's clearConsole(); function from console.h to clear the screen.

The playOneGame function (along with any sub-functions it calls within the same file) should perform **all** console user interaction, such as printing out the current state of the game. No other file should have **`cout`** or user/file input.

But boggleplay.cpp is not meant to be the place to store the majority of the game's state, logic, or algorithms. Your boggleplay file will interact with a class you will write named Boggle, described on the following pages. We describe a partial set of methods that your Boggle class must have. The intention is that your boggleplay code will call all of these methods to help achieve the overall task of playing the game. For example, no recursion or backtracking should take place in boggleplay; all such recursive searching should happen in the Boggle class. If you find that your boggleplay code is implementing a lot of complex logic itself, or that boggleplay is never calling a particular public method from the Boggle class, this is likely a sign that you have not divided the functionality in your code the way that we intend, which might lead to a style deduction.

Later in the spec we will describe a graphical user interface (GUI) that your Boggle game must display. As much as possible, the code to create and interact with this GUI should be in your boggleplay.cpp file. The one exception is the code to highlight and un-highlight letter cubes on the GUI while your algorithms are searching for words typed by the human player. Highlighting should be done in the Boggle class, because it would be very difficult to separate that code out of the recursive backtracking algorithms that are defined in the Boggle class.

Boggle.cpp / .h

The majority of your code should be in the Boggle.h and Boggle.cpp files, which should contain the implementation of a Boggle class. A Boggle object represents the current board and state for a single Boggle game, and it should have member functions to perform most major game functions like finding words on the board and keeping score. Declare all Boggle class members in Boggle.h, and implement their bodies in Boggle.cpp. We provide you a skeleton that declares some required members below that your class **must** have.

Do not change the headings of any of the following functions. Do not add parameters; do not rename them. You must implement exactly these functions with exactly these headings, or you will receive a deduction. (See, however, the note below about `const`.)

Boggle Method	Description
`Boggle(dictionary, boardText)`	In this constructor you initialize your Boggle board to use the given dictionary lexicon to look up words, and use the given 16-letter string to initialize the 16 board cubes from top-left to bottom-right. If the string is empty, you should generate a random shuffled board. Your method should be case-insensitive; it should accept the board text whether it is passed in upper, lower, or mixed case.
`b.getLetter(row, col)`	In this function you should return the character that is stored in your Boggle board at the given 0-based row and column. If the row and/or column are out of bounds, you should throw an int exception.
`b.checkWord(word)`	In this function you should check whether the given word string is suitable to search for on the board: that is, whether it is in the dictionary, long enough to be a valid Boggle word, and has not already been found. You should return a boolean result of true if the word is suitable, and otherwise you should return false. Your method should be case-insensitive; you should properly verify the word whether it is passed in upper, lower, or mixed case.
`b.humanWordSearch(word)`	In this function you should perform a search on the board for an individual word. If the word can be formed, you should add it to the human's set of found words. Your function returns a boolean result of whether the word can be formed. Your code for this function should use recursive backtracking. As each cube is explored, you should highlight it in the GUI to perform an animated search with a 100ms delay (see GUI page). If the word is unsuitable, you should not perform the recursive search. Your method should be case-insensitive; you should properly search for the word whether it is passed in upper, lower, or mixed case.
`b.computerWordSearch()`	In this function you should perform a search on the board for all words that can be formed (that have not already been found by the human player), and return them as a Set of strings. Your code for this function should use recursive backtracking. Though similar to your human search, this is different because you should look for all words and not perform any animation; therefore its code should be implemented separately from humanWordSearch. The words in your set should be all uppercase.

Boggle Method	Description
`b.getScoreHuman()`	In this function you return the total number of points the human player has scored in the game so far as an integer. This total is 0 when the game begins, but whenever a successful human word search is performed, points for that word are added to the human's total score.
`b.getScoreComputer()`	In this function you should return the total number of points the computer player has scored in the game as an integer. This total is initially 0 when the game begins, but after a computer word search is performed, all points for those words are added to the computer's total score.
`ostream& << b`	You should write a << operator for printing a Boggle object to the console. The operator should print only the 4x4 grid of characters representing the board as four lines of text. It should print the board text entirely in uppercase.

Once again, do not modify the names, parameters, or return types of the preceding functions; implement them as-is. The one exception to this rule is that you can (and should) modify headers to make the member function `const` if this function does not modify the state of your Boggle object. Review all of your functions (the ones provided above, and any others you choose to add to your class) and, whenever possible, make them const.

Case sensitivity: Your methods that accept strings must be case-insensitive; they should work with upper, lower, or mixed case. This should be enforced in your program by the Boggle class, not by the boggleplay.cpp code.

Adding your own member functions: In some past assignments, we gave you an exact list of the functions to implement. In this assignment, we are asking you to come up with some of the member functions. The Boggle class members listed on the previous page represent a large fraction of that class's behavior. But you can, and should, add other members to implement all of the appropriate behavior for this assignment. Your added members should be public if they are to be called directly by the boggleplay.cpp code, and private otherwise. You must also decide which code and/or data should go in boggleplay.cpp, and which should go in the Boggle class. Part of the challenge of this assignment is learning how to design a class and console UI client effectively. Remember that each member function of your class should have a single, clear, coherent purpose.

Here are some suggestions for good member functions to put in your Boggle class:

- Though the boggleplay.cpp file should do all console I/O, your Boggle class should have lots of convenient functions for boggleplay to call so that it doesn't need as much complex logic. No recursion or backtracking should take place in boggleplay; all such recursive searching should happen in the Boggle class.
- The boggleplay code needs to be able to display various aspects of the game state, such as all words that have been found by the each player, along with the players' scores. The Boggle class should keep track of such things, NOT boggleplay. The boggleplay code should ask the Boggle class for this information by calling accessory functions on it, which should return

appropriate data to the caller. Note that the Boggle class itself should not contain any output statements to `cout`; let boggleplay do that.

- Make a member function and/or parameter const if it does not perform modification of the object's state.
- Make a member function private if it is used internally and not to be called by the client (a "helper").
- Do not add functions to your Boggle class that directly return internal data structures in a way that allows the client to make direct modifications to them. (This is called "representation exposure" and is considered poor style.)

Member variables: We also have not specified any of the private member variables that should go inside the Boggle class; you must decide those yourself. Here are some thoughts about data members that your class might need:

- You'll certainly need a data structure to represent the current game board state, meaning the 16 letter cubes and which letter is showing on top of each cube. The exact choice of data structure is up to you, but you should make an efficient and appropriate choice from the Stanford libraries.
- It is fine to declare additional data structures, such as a collection of words found, etc.
- Don't make something a private data member if it is only needed by one function; make it local. Making a variable into a data member that could have been a local variable or parameter will hurt your Style grade.
- All data member variables inside a Boggle object should be private.

Searching: You don't want to visit the same letter cube twice during a given exploration, so for the search algorithm to work, your Boggle class needs some way to "mark" whether a letter cube has been visited or not. You could use a separate structure for marking, or modify your existing board, etc. It's up to you, as long as it is efficient and works.

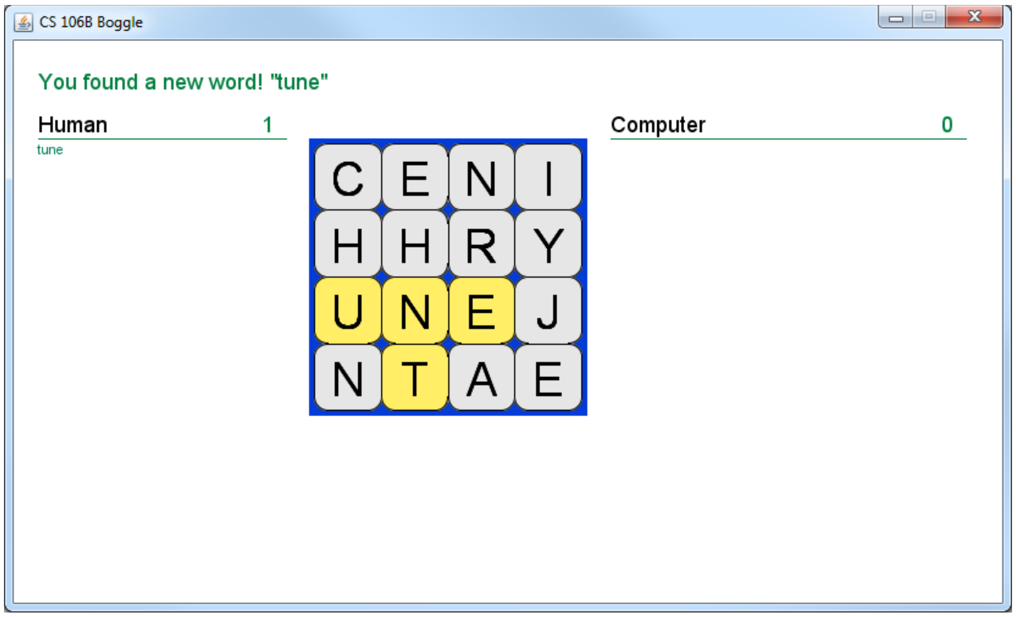
Please note that efficiency is very important for the part of your program that performs this search. It is important to limit your search to ensure that the process can be completed quickly. If written properly, the code to find all words on the board should run in around one second or less. To make sure your code is efficient enough, you must perform the following optimizations:

- Use a Lexicon data structure to store the English dictionary, and do not needlessly copy the Lexicon.
- Prune the tree of searches by not exploring partial paths that will be unable to form a valid word.
- Use efficient data structures elsewhere in your program (e.g. to represent which words are already found).

This second optimization, namely pruning dead-end searches, is one of the most important Boggle strategies. The Lexicon has a `containsPrefix` function that accepts a string and returns true if any word in the dictionary begins with that substring. For example, if the first cube you examine shows the letter Z and your algorithm tries to explore one of its neighbors that shows an X, your path would start with ZX. In this case, `containsPrefix` will inform you that there are no English words that begin with the prefix "ZX". Therefore your algorithm should stop exploring that path and move on to other combinations.

Graphical User Interface

As a required part of this assignment, you must also add a graphical user interface (GUI) to your program.



The GUI does not replace the console UI; it can't be clicked on to play the game, for example. It just shows a display of the current game state. To use the GUI, include bogglegui.h in your code, then call the functions below:

GUI Function	Description
`clearHighlight()`	Sets all letter cubes to be un-highlighted. (See setHighlighted.)
`initialize(rows, cols)`	Starts up the GUI and displays the graphical window. The board is drawn with empty squares and scores are 0. If called again, resets the board (see reset). Throws an error if rows or cols is not a positive integer from 1-6.
`isInitialized()`	Returns true if initialize has already been called.
`labelCube(r, c, char, bool)`	Sets the given cube to display the given character. Rows and columns have 0-based indexes with (0, 0) at top-left. If true is passed for the optional highlighted parameter, the cube is drawn with a colored highlight (useful to show progress of word searches). The highlight will remain until turned off. Throws an error if ch is not a letter or space.

GUI Function	**Description**
`labelAllCubes(str)`	Sets all letter cubes to display the characters from the given string. For example, "ABCDEFGHIJKLMNOP" would label cube (0, 0) with 'A', cube (0, 1) with 'B', and so on. The string can contain other characters such as whitespace, line breaks, etc., which will be skipped over. All cubes are un-highlighted after a call to this function. Throws an error if str does not contain 16 alphabetic letters.
`recordWord(word, player)`	Displays that the given player has found the given word string on the board. This function does not check word validity, e.g. that the word is not already shown, can be formed on the board, is in the dictionary, etc. That is up to you. The player parameter indicates which player found the given word; the value you pass should be either BoggleGUI::HUMAN or BoggleGUI::COMPUTER.
`reset()`	Sets the GUI window back to its initial state, with the letter cubes blank and unhighlighted, the scores both at 0, and no solved words shown on the screen.
`setAnimationDelay(ms)`	Sets a pause/delay of the given number of milliseconds. After calling this, subsequent calls to setHighlighted or to labelCube that have highlight set to true will trigger a pause. Use this to animate a word search algorithm.
`setHighlighted(r, c, bool)`	Sets the given letter cube to be highlighted (true) or un-highlighted (false).
`setScore(score, player)`	Sets the GUI's score display for the given player to the given number. The GUI does not know anything about scoring rules for Boggle; it accepts any integer. The player parameter indicates which player found the given word. The value you pass should be either BoggleGUI::HUMAN or BoggleGUI::COMPUTER .
`setStatusMessage(str)`	Displays a status message in the bottom part of the window. Useful for showing messages such as telling the user that they have found a word, etc.
`shutdown()`	Closes the GUI window and frees memory associated with the GUI.

The functions of the GUI are enclosed in a namespace so that they do not conflict with any other global function names in your program. To call one of them, you must prefix the function's name with BoggleGUI:: , like so:

```
BoggleGUI::recordWord("hello", BoggleGUI::HUMAN); // records the word "hello" for human player
```

You must call the GUI's setStatusMessage function to display information about the game state during play. Messages like "It's your turn!", "You must enter an unfound word ...", "That word can't be formed", "You found a new word", "It's my turn", "You defeated me", and "Ha ha ha, I destroyed you" should be shown. These are the same messages that display at the top of the text console on each turn. See the runnable sample solution for more details.

Development Strategy and Hints

In a project of this complexity, it is important that you get an early start and work consistently toward your goal. To be sure that you're making progress, it also helps to divide up the work into manageable pieces, each of which has identifiable milestones. Here's a suggested plan of attack that breaks the problem down into six phases:

- **Task 1**: Cube setup, board drawing, cube shaking. Design your data structure for the cubes and board. As usual, no global variables. Set up and shuffle the cubes. Use the provided shuffle function and/or the randomInteger function from random.h to help you make random choices. Add an option for the user to force the board configuration, as illustrated by the sample application.
- **Task 2**: Human's turn (except for finding words on the board). Write the loop that allows the user to enter words. Reject words that have already been entered, don't meet the minimum word length, or aren't in the lexicon. Don't worry about the recursive backtracking algorithm yet for verifying that the word can be formed from the cubes on the board; just perform the other validity checks and see if the word passes all of them.
- **Task 3**: Human's backtracking algorithm to find a given word on the board. Now use recursion to verify that a word can be formed on the board, subject to the various rules. You will employ recursive backtracking that "fails fast": as soon as you realize you can't form the word starting at a position, you backtrack.
- **Task 4**: Computer's turn (find all the words on the board). Now implement the killer computer player. Employing the power of recursion, your computer player traverses the board using an exhaustive search to find all remaining words. Be sure to use the lexicon prefix search to abandon searches down dead-end paths. NOTE: The program contains two recursive searches: one to find a specific word entered by the human player, and another to search the board exhaustively for the computer's turn. You might think that you should try to integrate the two into one combined function, by doing all word-finding at the beginning of the game, just after the board is initialized. But for full credit, you must implement the human and computer player as two separate search functions. There are enough differences between the two that they don't combine cleanly and the unified code is usually worse as a result. Focus on writing clean code that clearly communicates its algorithm.
- **Task 5**: Loop to play many games and add polish. Once you can successfully play one game, it's a snap to play many. Be sure to gracefully handle all user input so that it is not possible to break or crash the program.

- **Task 6**: Add the graphical user interface and animation. The GUI serves as a supplement to the existing text UI, not a replacement. So your text UI should still work properly in the presence of the GUI.

Make sure to extensively test your program. Run the sample solution (top of the page) to see the expected behavior of your program. When in doubt, match the behavior of the sample solution.

Possible Extra Features:

Congratulations—that's all! You are done. Consider adding extra features.

Though your solution to this assignment must match all of the specifications mentioned previously, you are allowed and encouraged to add extra features to your program if you'd like to go beyond the basic assignment. Here are some example ideas for extra features that you could add to your program.

1. **Make the Q a useful letter:** The Q is largely useless unless it is adjacent to a U, so the real Boggle prints Qu together on a single face of the cube. You use both letters together, a strategy that not only makes the Q more playable but also allows you to increase your score because the combination counts as two letters.
2. **Big Boggle:** Once you have a working program, it should require only a few changes to support a variant that uses a 5 × 5 board. Word game aficionados generally agree that the original size was just a bit too small and scaling it up adds to the fun and challenge. This is a great exercise in verifying that your design is sufficiently organized and flexible to permit this adaptation. Our starting code declares two different cube arrays, one with the 16 cubes for the standard game and another with the 25 cubes for the bigger version.
3. **Embellish the GUI:** Our Boggle GUI module is supplied in source form so you can adapt into a snazzier interface. For example, the current game merely highlights the word; it might be nice if it also drew lines or arrows marking the connections. Or you could use the Stanford C++ library's gevent.h facilities to let the user assemble a word by clicking or dragging the mouse through the letter cubes. Make it play sounds. Etc.
4. **Board exploration:** As you will learn, some Boggle boards are a lot more fruitful that others. Write some code to discover things about the possible boards. Is there an arrangement of the standard cubes that produces a board containing no words? What about an arrangement that produces a longest word, maybe even using all the cubes? What is the highest-scoring board you can construct? Recursion will be handy in trying out all the possible arrangements, but there are a lot of options (do the math on all the permutations...), so you may need to come up with some heuristics to direct your explorations.

Indicating that you have done extra features:

If you complete any extra features, then in the comment heading on the top of your program, please list all extra features that you worked on and where in the code they can be found (what functions, lines, etc. so that the grader can easily identify this code).

Submitting a program with extra features:

Since we use automated testing for part of our grading process, it is important that you submit a program that conforms to the preceding spec, even if you want to do extra features. If your feature(s) cause your program to change the output that it produces in such a way that it no longer matches the expected sample output test cases provided, you should submit two versions of your program file: a first one named life.cpp without any extra features added (or with all necessary features disabled or commented out), and a second one named life-extra.cpp with the extra features enabled. Please distinguish them in by explaining which is which in the comment header. Our submission system saves every submission you make, so if you make more than one we will be able to view all of them; your previously submitted files will not be lost or overwritten.

Suggested Milestones:

1. **On Wednesday, July 19:** Read the assignment, and understand the rules for Boggle.
2. **On Wednsday, July 19:** Read Chapter 6 in the book, on classes (covered in class on Tuesday)
3. **By Thursday, July 20:** Figure out how to set up the board, with the characters shuffled, or the input from the user directly.
4. **By Friday, July 21:** Implement the boggleplay.cpp functions, which set up the game and start the play.
5. **By Saturday, July 22:** Write the simple public functions in Boggle.cpp (e.g., getLetter(), checkWord(), getScoreHuman(), getScoreComputer()
6. **By Monday, July 24:** Write the humanWordSearch() function.
7. **By Tuesday, July 25:** Write the computerWordSearch() function.
8. **By Wednesday, July 26**: finish debugging and submit