

Stanford University, CS 106B, Autumn 2013
Homework Assignment 5: Boggle
due Friday, November 15, 2013, 2:00pm

Thanks to Julie Zelenski and Eric Roberts for creating this assignment.

This assignment focuses on recursive backtracking. Turn in the following files:

- **Boggle.h** / **.cpp** : files for a **Boggle** class representing the state of the Boggle game
- **boggleplay.cpp** : client to perform console UI and work with your **Boggle** class to play a game

We provide you with several other files to help you with the console UI, but you should not modify them.

The Game of Boggle:

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 are next to each other horizontally, vertically, or diagonally. There are up to eight letters near a cube. 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.

Your assignment is to write a program that plays a graphical rendition of this game, adapted for a single human player to play against a single computer player opponent. Unfortunately, the computer knows how to use **recursive backtracking**, so it is able to find every possible word on the board and destroy you every time. But it's still fun to write a program that can so soundly thrash 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:

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

Once the player has found as many words as he or 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 it finds *all* words.

Your program's output format should match the partial log of execution at right. See the course web site for complete example output files. Your program should exactly reproduce this output format and behavior.

```
(intro text trimmed)
```

```
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 press Enter
              to end your turn): room
```

```
You found a new word! "ROOM"
...
```

```
Your words (5): {"FOIL", "FORM",
                  "ROOF", "ROOM", "ROOMY"}
```

```
Your score: 6
```

```
Type a word (or press Enter
              to end your turn):
```

```
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!
```

```
Play again (Y/N)? n
Have a nice day.
```

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 lists all of the letters on all six faces of each of the sixteen cubes from the original Boggle. 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:

- 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.)
- 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 libraries do not contain a function to **shuffle**, but for this assignment we include a file [shuffle.h](#) that you can include with a **shuffle** function you can use to randomly rearrange the elements of an array, **Vector**, or **Grid**. In case you are curious about how shuffling works, the algorithm follows this pseudo-code:

```
function shuffle(List):  
    for each int i from 0 to List's size - 1:  
        Choose a random index r between i and the last element index, inclusive.  
        Swap the elements at positions i and r.
```

Your game must also have an option where the user can enter a **manual board configuration**. In this option, rather than randomly choosing the letters to 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 should not 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 described previously, 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 invalid words also do not 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: a 4-letter word is worth 1 point; a 5-letter word is worth 2 points; 6-letter words are worth 3; and so on. The player enters a blank line when 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 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 is roughly the following:

```

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

```

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

Efficiency is very important for this part of the program. As with any exponential search algorithm, it is important to look for ways to limit the search to ensure that the process can be completed in a reasonable time. 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
- prune the tree of searches by not exploring partial paths that will be unable to form a valid word
- use efficient data structures otherwise in your program (*e.g. to represent which words are already found*)

One of the most important Boggle strategies is to **prune dead-end searches**. The **Lexicon** has a **containsPrefix** member function that accepts a string parameter and returns **true** if any word in the dictionary begins with that substring. For example, if the first cube you choose to start from is showing the letter **Z** and then your algorithm tries to explore one of its neighbors that shows an **X**, you will have a path starting with **ZX**. In this case, the **containsPrefix** function will inform you that there are no English words that begin with the prefix **"ZX"**. Therefore your algorithm should stop that path and move on to more promising combinations.

Implementation Details:

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 already written; 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 provided **yesOrNo** function from **bogglemain.cpp** that prompts the user to type yes or 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 such as the board words found so far, score, etc. This makes a more pleasant UI where 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 should perform all console user interaction such as printing out the current state of the game. This is the *only* file in which you should have any statements that read/write to **cout** or **cin**. But **boggleplay.cpp** is *not* meant to be the place to store the majority of the game's state, logic, or algorithms. In fact, our reference solution for this file is only around 60 lines long including comments. Instead, the majority of your code should be in the **Boggle.h** / **.cpp** files, which should contain the implementation of a **Boggle** class. A **Boggle** object represents the current game board and game state, and it should have member functions to perform most of the major game functions like finding words on the board, keeping score, and determining who has won the game. Declare all **Boggle** class members in **Boggle.h**, and implement their bodies in **Boggle.cpp**.

In past assignments, we gave you an exact list of what members you were required to implement. In this assignment, we are asking you to come up with the members. That is part of the challenge of this assignment: learning how to design a class and its console UI client effectively. Remember that each member function of your class should have a clear, coherent purpose. You must also decide the private data members that should go inside of a **Boggle** object.

Here are some thoughts about **private data members** that you might need in your **Boggle** class:

- You'll certainly need a data structure to represent the current **game board state**, meaning the 16 letter cubes and what 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.
- You should also store a **Lexicon** inside your boggle game representing the English **dictionary**. We provide a

resource file **EnglishWords.dat** containing over 125,000 words that you must use to initialize your **Lexicon**. This file is in an efficient binary format that will optimize the **Lexicon** for searching for words and prefixes.

- It is fine to declare **additional data structures**, such as a collection of the words found on the board, etc.
- Don't make something a **private** data member if it is only needed by one member function. Make it local.
- All data member variables inside a **Boggle** object should be **private**, unless they are **const** global constants.

Here are some suggestions for good **public member functions** that you might need 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 it to call so that it doesn't need to have any complicated logic. For example, write functions where **boggleplay** can pass a word to your **Boggle** class and ask various questions about it, such as if it is long enough, if it is found in the dictionary, if it has already been played, and so on. The **boggleplay** client can call these, and depending on the result that is returned, it can print an appropriate message.
- The **boggleplay** code needs to be able to display all words that have been found by the human player, along with the player's score. The **Boggle** class should keep track of such things, NOT **boggleplay**. The **boggleplay** code should ask the **Boggle** class for this information which should be returned or sent out to it. Note that the **Boggle** class itself should not contain any output statements to **cout**; let **boggleplay** do that.
- Each time the user says he/she wants to play a new game, you'll need to reset the state of the game, clear out any words previously found, reset the score, and so on. The **Boggle** class should contain all of the logic required to reset this state, as well as the code to roll the letter cubes and shuffle up the 16 letters into a new combination at the start of each game. Remember that your Boggle game should also contain a way for the **boggleplay** client to initialize a new game using a fixed 16-letter string representing the letters the user wants to have on the board.
- Make a function and/or parameter **const** if it does not perform modification of object state.
- Make a function **private** if it is only used internally and not meant to be called by the client (*e.g. a helper*).

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 five 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:** 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.

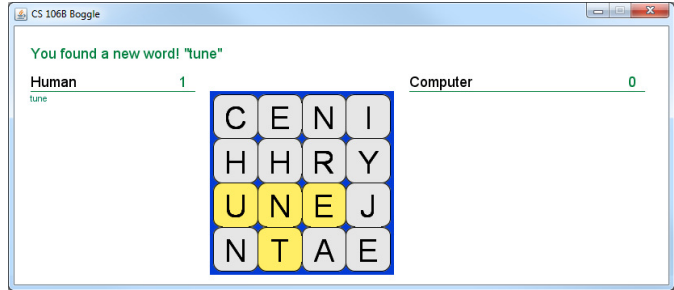
NOTE: The program contains two recursive searches: one to find a specific word entered by the human player, and another to search the board exhaustively for the computer's turn. You might think that you should try to integrate the two into one combined function, by doing all word-finding at the beginning 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 made 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.

Graphical User Interface (optional):

We have provided some files if you would like to add an optional 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 board, words, and state.

If you want the GUI, include `bogglegui.h` in your source code. It declares the following graphical functions (*which you can call from any of your code files*):



<code>initialize(rows, cols)</code>	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 <code>reset</code>). Throws an error if rows or cols is not a positive integer from 1-6.
<code>isInitialized()</code>	Returns true if <code>initialize</code> has already been called.
<code>labelCube(row, col, ch, highlight)</code>	Sets the given letter cube to display the given character. Rows and columns have 0-based indexes starting with (0, 0) at top-left. If true is passed for the optional highlight parameter, the cube is drawn with a colored highlight (useful to show progress of word searches). The highlight will remain until turned off. Letters can be passed in either upper-or-lowercase but will always be displayed in uppercase. Throws an error if ch is not a letter or space.
<code>labelAllCubes(str)</code>	Sets all letter cubes to display the characters from the given string. For example, passing "ABCDEFGHJKLMNOP" 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.
<code>recordWord(word, player)</code>	Displays that the given player has found the given word string on the board. Also increments the player's word count by 1 on the screen. This function does not check word validity, e.g. that the word is not already shown, that it can be formed on the current board, is long enough, 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 <code>BoggleGUI::HUMAN</code> or <code>BoggleGUI::COMPUTER</code> .
<code>reset()</code>	Sets the GUI window back to its initial state, with the letter cubes blank and un-highlighted, the scores both at 0, and no solved words shown on the screen.
<code>setAnimationDelay(ms)</code>	Sets a pause/delay of the given number of milliseconds. After calling this, subsequent calls to <code>labelCube</code> that have highlight set to true will trigger a pause. This is useful for animating a word search algorithm.
<code>setScore(score, player)</code>	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 will accept any integer. The player parameter indicates which player found the given word. The value you pass should be either <code>BoggleGUI::HUMAN</code> or <code>BoggleGUI::COMPUTER</code> .
<code>setStatusMessage(str)</code>	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.
<code>shutdown()</code>	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::`, such as:

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


Style Guidelines and Grading:

In general, items mentioned in the "Implementation and Grading" from the previous assignment(s) specs also apply here. Please refer to those documents as needed. Note the instructions in the previous assignments about procedural decomposition, variables, types, parameters, value vs. reference, and commenting. Don't forget to **cite any sources** you used in your comments. Refer to the course **Style Guide** for a more thorough discussion of good coding style.

Part of your grade will come from appropriately utilizing recursive backtracking to implement your word-finding algorithm as described previously. We will also grade on the elegance of your recursive algorithm; don't create special cases in your recursive code if they are not necessary. Redundancy is another major grading focus; avoid repeated logic as much as possible. Your **Boggle** class will be graded on whether you make good choices about what members it should have, and other factors such as which are **public** vs. **private**, and **const**-correctness, and so on.

Your code should have no **memory leaks**. Don't use pointers or **new** and you won't need to explicitly free anything.

As for **commenting**, place a descriptive comment header on each file you submit. In your **.h** files, place detailed comment headers next to every member explaining its purpose, parameters, what it returns, any exceptions it throws, assumptions it makes, etc. You don't need to put any comment headers on those same members in the corresponding **.cpp** file, though you should place inline comments as needed on any complex code in the bodies.

For reference, our solution to the **Boggle** class has 12 public member functions, 4 private member variables (fields), and a few private helper member functions. Our **Boggle.cpp** is around 160 lines not including comments, and our **boggleplay.cpp** is around 80 lines including comments. You don't have to match these numbers or even come close to them; they are just to use as a reference and a sanity check.

Please remember to follow the **Honor Code** on this assignment. Submit your own work; do not look at others' solutions. Cite sources. Do not give out your solution; do not place a solution on a public web site or forum.

Copyright © Stanford University and Marty Stepp, licensed under Creative Commons Attribution 2.5 License. All rights reserved.