Assignment 3: Boggle
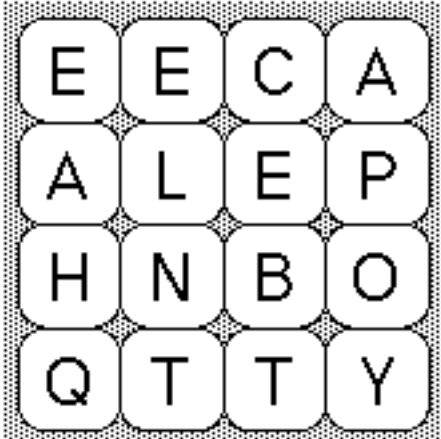
Thanks to Todd Feldman for the original idea behind the Boggle assignment.

Me   9

lean	peel	clean
pace	pent	lent
bent	clan	

Computer   56

elan	celeb	cape	capelan	capo
cent	cento	alee	alec	anele
leant	lane	leap	lento	peace
pele	penal	hale	hant	neap
bleep	blae	blah	blent	becap
benthal	bott	open	thae	than
thane	toecap	toea	tope	topee
toby				

Boggle board:

E	E	C	A
A	L	E	P
H	N	B	O
Q	T	T	Y

The Game of Boggle

Those of you fortunate enough to have spent summers seeing the world from the back of the family station wagon with the 'rents and sibs may be familiar with Boggle, the little word game that travels so well, and those who didn't will soon become acquainted with this vocabulary-building favorite. The Boggle board is a 4x4 grid onto which you shake and randomly distribute 16 dice. These 6-sided dice have letters rather than numbers on the faces, creating a grid of letters from which you can form words. In the original version, the players all start together and write down all the words they can find by tracing by a path through adjoining letters. Two letters adjoin if they are next to each other horizontally, vertically, or diagonally. There are up to eight letters adjoining a cube. A letter can only be used once in the word. When time is called, duplicates are removed from the players' lists and the players receive points for their remaining words based on the word lengths.

Solutions to Warm-up Problems Due: Monday, April 26th at 5:00 p.m.
Solution to Boggle Due: Friday, April 30th at 5:00 p.m.

This part of Assignment 3 is to write a program that plays a fun, graphical rendition of this little charmer, adapted to allow the human and machine to play one another. As you can see from the screen shot above, the computer basically trounces all over you, but it's fun to play anyway.

The main focus of this part of the assignment is designing and implementing the recursive algorithms required to find and verify words that appear in the Boggle board. This problem is larger than either of the two problems appearing in Handout 19, so don't be lulled into thinking it can be done in one sitting.

How's this going to work?

You will read the letter cubes in from a file and shake the cubes up and lay them out on the board graphically. The human player gets to go first (nothing like trying to give yourself the advantage). The player proceeds to enter, one at a time, each word that she finds. Your program is to verify that the word meets the minimum length requirement (which is 4), has not been guessed before, is a legal word in the English language, and can, in fact, be formed from the dice on the board. If so, the letters forming the word are highlighted, the word is added to the player's word list, and she is awarded points according to the word's length.

The player indicates that she is done entering words by hitting a lone extra carriage return. At this point, the computer gets to take a turn. The computer player searches through the board looking for words that the player didn't find and award itself points for finding all of them. The computer typically beats the player mercilessly, but the player is free to try again, you should play as many games as you want before exiting. Each time you repeat this entire process.

The Dice

The letters in Boggle are not simply chosen at random. Instead, the letters on the faces of the cubes are arranged in such a way that common letters come up more often and it is easier to get a good mix of vowels and consonants. To recreate this, we give you a text file that contains descriptions of the actual sixteen dice used in Boggle. (We've also included a file for the 5 x 5 version of Boggle, but that's completely optional and you don't need to build the 5 x 5 version if you don't want to.) Each die description is a single line of 6 letters; they are *not* separated by spaces. For example, the 4 x 4 file might look something like this:

```
EDAIJW
KZBEDT
ULNEEP
// 13 more lines follow this one
```

During initialization, you read the dice file and store it into a suitable data structure for subsequent use. At the beginning of each game, you "shake" the dice to randomly set-up the board. There are two different random aspects to consider. First, the cubes themselves need to be shuffled so that the same die is not always in the same cell of the board. Second, a random side from each die needs to be chosen to be the face-up letter.

Choosing a random side is a straightforward use of the random library. Shuffling is slightly more involved, and complicated a bit because of the two-dimensional nature of the data structure. To re-arrange the cubes themselves, you can use the simple shuffling algorithm given by this pseudo-code to mix up the elements in a two dimensional grid:

```
for (int lh = 0; lh < board.numRows * board.numCols(); lh++) {
   int rh = RandomInteger(lh, board.numRows * board.numCols() — 1);
   swap(board[lh / board.numCols()][lh % board.numCols],
        board[rh / board.numCols()][rh % board.numCols]);
}
```

Basically, this walks over the grid selecting a random element to place in each slot. Put something like this together with a way to select the side to put up in each position and you should be able to shuffle the dice into many different board combinations.

Alternatively, the user may decide to enter his or her own board configuration. In this case, you can still use your same data structure. The only difference is where the letters come from. If the user wishes to enter in a custom board, you should prompt them for a string of sixteen characters, representing the cubes from left to right, top to bottom (as you would read a book). You should verify that this string is at least sixteen characters and re-prompt if it is too short. If it's longer than sixteen characters, just ignore those characters after the ones you need. You don't need to verify that the characters are legal letters. Once your initialization is complete, you're ready to implement the two types of recursive search: one for the user, and one for the computer.

There are two distinct types of recursion happening here. For the user, you search for a specific word and stop as soon as you find it. For the computer, you are searching for all words. While you may be tempted to try and integrate the two so they work as a single type of recursion, this is not a good idea for two reasons. One is that your program will get very messy trying to integrate the two, as they are not algorithms that can be unified well. The other is that we want you to get practice with both types. Because of this, **we explicitly require that you implement two separate recursive functions**, one for the human player's turn (searching for a specific word) and for the computer's turn (exhaustive search for all words).

The human player's turn

After the board is displayed, the player gets a chance to enter all the words she can find on the board. Your program must read in a list of words until the user signals the end of the list by typing a blank line. As the user enters each word, your program must check the following conditions:

- That the word is at least four letters long.
- That it is defined in the lexicon as a legal word.
- That it occurs legally on the board (i.e., it is composed of adjoining letters such that no board position is used more than once).
- That it has not already been included in the user's word list.

If any of these conditions fail, you should tell the user about it and not give any score for the word. If, however, the word satisfies all these conditions, you should add it to the user's word list and update their score appropriately. In addition, you should use the facilities provided by the **gboggle.h** interface to highlight the word. Because you don't want the highlight to remain on the screen indefinitely, you should highlight the letters in the word, pause for about a second using the **Pause** function in the extended graphics library, and then go back and remove the highlights from the letters in the word.

Word length determines point value: 1 point for the word itself and 1 additional point for every letter over the minimum. Since the minimum word length is 4, **"boot"** gets 1 point, **"smack"** gets 2, and **"frazzled"** gets 5. The functions in the **gboggle** module will help you to display the player word lists and track the scoring. The player enters a lone carriage return (blank line) when done entering words.

The computer's turn

On the computer's turn, your job is to find all of the words that the human player missed by recursively searching the board for words beginning at each square on the board. In this phase, the same conditions apply as on the user's turn, plus the additional restriction that the computer is not allowed to count any of the words already found.

As with any exponential search algorithm, it is important to limit the search as much as you can to ensure that the process can be completed in a reasonable time. One of the most important strategies is to recognize when you are going down a dead end so you can abandon it. For example, if you have built a path starting with the prefix **"zx"**, you can use the lexicon's **containsPrefix** method to determine that there are no English words beginning with that prefix. So, you can stop right there and move on to more promising combinations. If you miss this optimization, you'll find yourself taking long coffee breaks while the computer is busy checking out non-existent words like **"zxgub"**, **"zxaep"**, etc. Not what you want.

The `gboggle` module

As mentioned before, we have written all the fancy graphics functions for you. The functions from the `gboggle.h` interface are used to manage the appearance of the game on the display screen. It includes functions for initializing the display, labeling the cubes with letters, highlighting cubes to indicate that they are part of a word, and displaying words found by each player. Read the interface file (in the starter folder) to learn how to use the exported functions. The implementation is provided to you in source form so you can extend this code in your own novel ways.

Solution strategies

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 the following five phases:

- *Task 1—Dice reading, board drawing, dice shaking.* Design your data structure for the dice and board. It will help to group related data into sensible structures rather than pass a dozen parameters around. **You should <u>not</u> use any global variables in this program.** Read the dice file and store the dice. Create your shuffling routine. Use the `gboggle` routines to draw the starting board. Add an option for the user to specify a particular board configuration.
- *Task 2—User's turn (except for finding words on the board).* Write the loop that allows the user to enter her words. Reject words that have already been entered or that don't meet the minimum word length or that aren't in the lexicon. Do not assume there is any upper limit on the number of words that may be found by the user. Put the `gboggle` functions to work for you adding words to the graphical display and keeping score. At this point, the words the user enters may or may not be possible to form on the board, that's coming up next.
- *Task 3—Find a given word on the board.* Now you will go to test your recursive talents in verifying that the user's words can actually be formed from the board. Remember that a valid word must obey the neighbor and non-duplication rules. You should search the board recursively, trying to find a legal formation of the user's word. This recursion is what you might call a "fail-fast" recursion, as soon as you realize you can't form the word starting at a position, you need to move on to the next position. Reject any word that cannot be formed from the letters currently on the board. Use the highlighting function from `gboggle` to temporarily draw attention to the letters in the word once you have verified it can be formed on the board.
- *Task 4—Find all the words on the board (computer's turn).* Now it's time to implement the killer computer player. Your computer player will make mincemeat of the paltry human player by traversing the board and finding every word the user missed. This recursion is an exhaustive search, so you will completely explore all positions on the board hunting for possible words. This phase is where the most

difficult applications of recursion come into play. It is easy to get lost in the recursive decomposition and you should think carefully about how to proceed.

- *Task 5—Loop to play many games, add polish.* Once you can successfully play one game, it's a snap to allow the user can play as many games as she likes. Finish off the little details. Make sure you gracefully handle all user input. Add sounds, if you're up for it. Note that adding sounds is <u>not</u> a requirement for the assignment.

A little more challenge: fun extras and extension ideas

As with most assignments, Boggle has many opportunities for extension. The following list may give you some ideas but is in no sense definitive. Use your imagination!

- *Make the Q a useful letter.* Because the **Q** is largely useless unless it is adjacent to a **U**, the commercial version of Boggle prints **Qu** together on a single face of the cube. You get to 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.
- *Embellish the program with better graphics.* The current game merely highlights the word; the words might be clearer if it also drew lines or arrows marking the connections.
- *Use the mouse to trace the word on the board.* The extended graphics library allows you to read the location of the mouse and determine whether the button is pressed. You can use these functions to allow the user to assemble a word by clicking or dragging through the letter cubes.
- *Add sound.* Just for fun, there is a random smattering of sound files provided with the project you can sprinkle about to liven up the game. Our CS106 **sound.h** header file exports the function **PlayNamedSound**, which allows you to play a sound from a file. There is another exported function **SetSoundOn** that gives you a global control for enabling or disabling sound to avoid annoying your long-suffering roommate during your late night coding marathons. Take a look in the **Sounds** folder of the starter project to see the various sound files we've put together.

Accessing files

On the class web site, there are two folders of starter files: one for XCode and one for Visual Studio. Each folder contains these files:

boggle.cpp	Source file for your Boggle game implementation.
gboggle.cpp	Source file which implements the **gboggle** interface.
gboggle.h	Interface file for the **gboggle** module.
cubes16.dat	Data file containing the boggle cubes for a 4 x 4 board.
cubes25.dat	Data file containing the boggle cubes for a 5 x 5 board. (This one is completely optional.)
lexicon.dat	Data file containing a large word list for the lexicon in binary format.

Sounds   Directory of sound files
BoggleDemo   A working program that illustrates how the game is played.

Deliverables

As always, you are to submit both a printed version of your code in lecture, as well as an electronic version. Both are due by 5:00 p.m. on Friday (though solutions to the problems treated by Handout 18 are due the previous Monday by 5:00 p.m.). When submitting the electronic version, please submit only the source files you edited (most likely just **boggle.cpp**. We don't need library and sound files and so on, unless there are special extensions you've added to them).