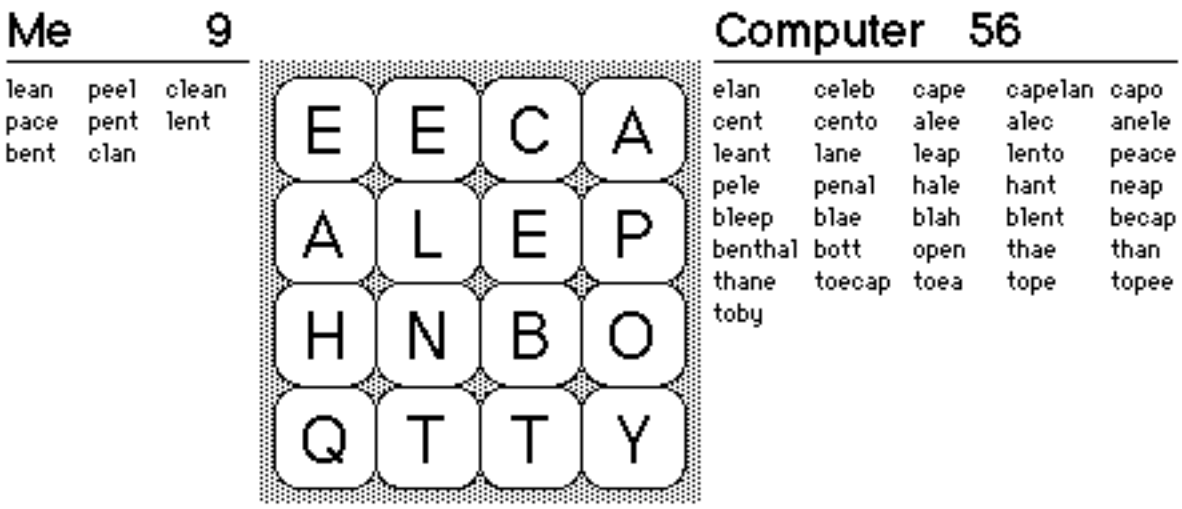
Assignment #5: Boggle™

Credit to Todd Feldman for the original idea for Boggle.



Due: Wed May 10th in class

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 on which you 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.

Your assignment is to write a program that plays a fun, graphical rendition of this little charmer, adapted to allow the human and machine to play pitted against 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.

This main focus of this assignment is designing and implementing the recursive algorithms required to find and verify words that appear in the Boggle board. Because we want you to concentrate on the strategy section, we have given you two modules to start with, the gboggle module which is responsible for the graphical display, and the lexicon module which is responsible for keeping track of a list of legal words (both of these are discussed more later). There is still a lot of work left for you, so definitely do not postpone this assignment until the night before.

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. If a word meets the minimum length requirement (which is 4), has not been guessed before, and can, in fact, be formed from the dice on the board, the letters forming the word are highlighted graphically (details to follow). Then the word is checked against the list of legal words in the lexicon. If it exists, the word is added to

the player's word list, and she is awarded points according to the word's length. If the word doesn't appear in the lexicon, the player is chided for trying to pull a fast one on the computer.

The player indicates that she is through 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 those words. The computer typically beats the player mercilessly, but the player is free to try again, you should play as many games as the user wants 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. Each die description is a single line of 6 letters; they are *not* separated by spaces. For example, an acceptable file could look something like this:

```
EDAIJW
KZBEDT
ULNEEP
. . .      /* 13 lines to follow */
```

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, on the other hand, is slightly more involved. 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 array:

```
for (row = 0; row < numRows; row++)
   for (col = 0; col < numCols; col++)
      swap elements array[row][col] and
      array[RandomInt(row, numRows-1)][RandomInt(col, numCols-1)]
```

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

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 score. 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.

A word's point value is determined by its length: 1 point for the word itself and 1 additional point for every letter over the minimum. Since the minimum word length is 4, the word "boot" receives 1 point, "trees" receives 2. The functions in the graphics module will help you to display the player word lists and track the scoring. The player enters a lone carriage return 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 here is to realize when you are going down a dead end so you can abandon it. For example, if you have built a path starting with "zx", you can use the lexicon's **`IsPrefix`** function to determine that there are no words in English that begin with that prefix. Thus, 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 looking up all those non-existent words like zxgub, zxaep, etc.

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 and re-printed at the end of this handout) to learn how to use the four exported functions. The implementation is provided to you in source form so you can extend this code if you're interested in pushing the limits of the assignment.

The lexiconADT module

You will need to verify the user's and computer's words by looking them in up in a lexicon. Although the word *lexicon* is often used as a synonym for *dictionary*, lexicon doesn't imply that the words have definitions in the way that dictionary entries do. Thus we will use the term lexicon since we only are keeping tracking of the words themselves, the definitions are superfluous. All you need to know is whether a word is in the list. The lexicon abstraction provides this capability.

The functions in this interface allow you to create a new lexicon and add words to it, either one at a time or as an entire collection stored in a data file. You can also determine whether a string matches a word in the lexicon or whether it is the prefix of some longer word. The complete interface file is included at the end of the handout for your perusal.

The lexicon is an example of an *abstract data type*, which is often abbreviated to *ADT*. We'll soon spend a lot of time talking about ADTs. . The beauty of an ADT is that you shouldn't need to know how it works to use it. You read the interface file and you'll see there are functions to create new lexicons, add words to them, check if words are defined, and so forth. The implementation of the lexicon abstraction is hidden from you and given as a precompiled library called **`lexicon.lib`**. Behind the scenes, the lexicon implementation uses an extremely efficient and compact yet very complex representation to give you blindingly fast access to a list of about 100, 000 words, but you don't need to know any of that in order to use it!

Solution strategies

In a project of this size, it is extremely 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 doing into the following five phases:

- *Task 1—Welcome, loop to play many games.* This is just straightforward simple code, so nothing too troublesome about getting this piece into place. "Playing a game" at this point doesn't mean much, but this is a good start nonetheless.
- *Task 2—Dice reading, board drawing, dice shaking.* Design your data structure for the dice and board. You should not 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.
- *Task 3—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 4—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 adjoinment 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 5—Find all the words on the board (computer's turn).* Now it's time to implement the killer computer player. With the power of recursion and a super-snappy and huge lexicon, 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, 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.

Other random notes and suggestions

- It's probably worth it to first spend some time with the demo program making sure you understand the rules and game-handling so you can be sure your version meets our requirements.
- As always, you want to be careful about keeping yourself with the array bounds. Unlike the GridADT from Life that conveniently reported when you went off the board, ordinary C arrays don't provide any sort of bounds-checking so you need to be extra-careful about watching for the boundary conditions. Funneling all your access through a helper that does bounds-check may be a worthwhile development aid.
- Using the `strlib.h` library tends to be wasteful of memory because functions like `Concat` dynamically allocate new memory each time they are called. If you don't free that memory, it will lie orphaned in the heap, and it may not be possible for you to play more than a few games of Boggle before you run out of memory. That's ok— for this program, we encourage you to ignore the issue of freeing memory since freeing things can be tricky and when you get it wrong, the punishment can be steep. Only once you have your entire program working, you might try to cleanup its memory usage. Concentrate on those strings created during the recursive calls, since there are so many of them. You can address the memory inefficiency either

by freeing the memory you no longer need or by avoiding the dynamic allocation in the first place (the latter can be an easier approach).

- As with Battleship, there is a random smattering of sounds you can sprinkle about to liven up the game and annoy the poor people working next to you. Supporting sound is not required, but it can be fun and it is easy to play prerecorded sounds. If you are using CodeWarrior, add the Macintosh **`sounds.rsrc`** file to your project, (just like a .c file). If you are using Visual C++, there are individual .wav files for each sound. You do not need to add the .wav files to your project, just make sure the files are in the same folder as your project. Then (in either environment), add **`#include "sound.h"`** to your code, and use the function **`PlayNamedSound`** to play sounds. For example, if you have a sound called "Whoops", you can play it using the function call **`PlayNamedSound("Whoops").`**

  The available sounds by name are: "come on faster", "not", "That's Pathetic", "Excellent", "Denied", "Dice Rattle", "yeah right", "whoops", "not fooling anyone", "oh really", tinkerbell", "tweetle", "idiot", "yah as if"
- You may need to increase the amount of heap memory for the Macintosh CW project to allow extra memory for the sounds. Set the heap size under the Edit->Project Settings->PPC Target to something like 2 or 3K if make enough space for the graphics, sounds, and lexicon to peacefully co-exist.

A little extra challenge: extension ideas

If you're somehow got some spare time to burn (ha!), 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.
- *Allow multiple human players.* Change the game so that both sides can be played by human players or, better yet, so that the game can have more than two players. With multiple human players, it is extremely difficult to support simultaneous turns (there is only one mouse and keyboard), but you can clear the screen for one player before passing it on to the next. This extension complicates the scoring because points cannot be recorded at the time the words are formed; all scoring must instead be deferred to the end of the game because points for a word are awarded only if that word appears on no one else's list.

Accessing Files

On the class web site, there are two folders of starter files: one for Mac CodeWarrior and one for Visual C++. Each folder contains these files:

gboggle.c	Source file which implements the gboggle interface.
gboggle.h	Interface file for the gboggle module.
lexicon.lib	Compiled library which implements the lexicon abstraction.
lexicon.h	Interface file for the lexicon abstraction.

cubes.dat	Data file containing the boggle cubes.
words.dat	Data file containing a large word list for the lexicon in binary format.
sounds.rsrc	Resource file containing all sounds (Mac)
not.wav, moo.wav	Individual sound files (PC)
BoggleDemo	A working program that illustrates how the game is played.

To get started, create your own starter project and add **`gboggle.c`** and **`lexicon.lib`** to it, as well as your own source code file, which you should call **`boggle.c`**. If you are using sounds on the Mac, you will also need to add **`sounds.rsrc`** to the project.

The demo version is a good place to start to make sure you understand how it operates. In order for the game to work, the data files (dice & dictionary) must be in the same folder as the application.

Deliverables

At the beginning of Wednesday's lecture, turn a manila envelope containing a printout of your code, and a floppy disk containing your entire project. Everything should be clearly marked with your name, CS106X and your section leader's name!

Did you hear about the computer scientist who died in the shower shampooing his hair? The instructions said "Lather, Rinse, Repeat" and he died of a stack-heap collision.

```
/* File: gboggle.h
 * ---------------
 * The gboggle.h file defines the interface for a set of functions that
 *   1. Draw the boggle board
 *   2. Manage the word lists
 *   3. Update the scoreboard
 */
#ifndef _gboggle_h
#define _gboggle_h

/* Type: playerT
 * -------------
 * This enumeration distinguishes the human and computer players.
 */
typedef enum {Human, Computer} playerT;

/* Function: DrawBoard
 * Usage: DrawBoard();
 * -------------------
 * This function draws the empty layout of the board.  It should be called once at
 * the beginning of each game, after calling InitGraphics() to erase the graphics
 * window.  It will redraw the boggle cubes, board, and scores.  It resets the scores
 * and word lists to zero for each player.  The boggle cubes are drawn with blank
 * faces, ready for letters to be set using the LabelCube function.
 */
void DrawBoard(void);


/* Function: LabelCube
 * Usage: LabelCube(row, col, letter);
 * -----------------------------------
 * This function draws the specified letter on the face of the cube at position (row,
 * col).  The cubes are numbered from top to bottom, left to right starting with zero.
 * Therefore, the upper left corner is (0, 0); the lower right is (3, 3).  Thus,
 * the call LabelCube(0, 3, 'D') would put a D in the top right corner cube.
 */
void LabelCube(int row, int col, char letter);


/* Function: HighlightCube
 * Usage: HighlightCube(row, col, flag);
 * -------------------------------------
 * This function highlights or unhighlights the specified cube according to the
 * setting of flag: if flag is TRUE, the cube is highlighted; if flag is FALSE,
 * the highlight is removed. The highlight flag makes it possible for you to show
 * which letters are in a word.
 */
void HighlightCube(int row, int col, bool flag);


/* Function: RecordNewWord
 * Usage: RecordNewWord(word, player);
 * -----------------------------------
 * This function records the specified word by adding it to the screen display for the
 * specified player and updating the scoreboard accordingly.  Scoring is calculated as
 * follows:  a 4-letter word is worth 1 point, a 5-letter is worth 2, and so on.
 */
void RecordNewWord(string word, playerT player);

#endif
```

```
/*
 * File: lexicon.h
 * ---------------
 * This file is the interface to an abstraction which allows the client to keep track
 * of a list of words.  The main difference between the lexicon abstraction and a
 * symbol table or dictionary is that the lexicon does not provide any mechanism for
 * storing the definitions; a string in a lexicon is either a word or it isn't.
 */

#ifndef _lexicon_h
#define _lexicon_h

#include "genlib.h"

/*
 * Type: lexiconADT
 * ----------------
 * This type represents the ADT for a lexicon.  The type is a purely abstract type
 * in this interface, defined entirely in terms of its operations. Because the type
 * is represented as a pointer to an incomplete structure type, the client has no
 * access to the underlying record structure.
 */
typedef struct lexiconCDT *lexiconADT;


/*
 * Function: NewLexicon
 * Usage: lexicon = NewLexicon();
 * ------------------------------
 * This function creates a new lexicon and initializes it to be empty.
 */
lexiconADT NewLexicon(void);


/*
 * Function: FreeLexicon
 * Usage: FreeLexicon(lexicon);
 * ----------------------------
 * This function frees the storage associated with the lexicon.
 */
void FreeLexicon(lexiconADT lexicon);


/*
 * Function: AddWordToLexicon
 * Usage: AddWordToLexicon(lexicon, word);
 * ---------------------------------------
 * This function adds the specified word to the lexicon.
 */
void AddWordToLexicon(lexiconADT lexicon, string word);


/*
 * Function: ReadLexiconFile
 * Usage:  ReadLexiconFile(lexicon, filename);
 * -------------------------------------------
 * Opens the specified file, expecting a binary format that represents a
 * large lexicon more efficiently and adds all of the words found. The file must
 * be in the same folder as the project to be found. If file doesn't exist or
```

```
 * memory is exhausted before constructing the lexicon, this function will call
 * Error to exit the program.
 */
void ReadLexiconFile(lexiconADT lexicon, string filename);

/*
 * Function: WordsInLexicon
 * Usage: n = WordsInLexicon(lexicon);
 * -----------------------------------
 * This function returns the number of words in the lexicon.
 */
long WordsInLexicon(lexiconADT lexicon);

/*
 * Function: IsWord
 * Usage: flag = IsWord(lexicon, word);
 * ------------------------------------
 * This function looks up the specified word in the lexicon and returns
 * TRUE if this word is a valid entry.
 */
bool IsWord(lexiconADT lexicon, string word);


/*
 * Function: IsPrefix
 * Usage: flag = IsPrefix(lexicon, word);
 * --------------------------------------
 * This function returns TRUE if any words in the lexicon begin with the
 * specified prefix.  A word is defined to be a prefix of itself and the
 * empty string is a prefix of everything.
 */
bool IsPrefix(lexiconADT lexicon, string prefix);



#endif
```