

## Assignment #6 - Boggle

**Due: Friday, January 28, at 5:00 pm**

This assignment, which was originally developed by Todd Feldman and then enhanced by Julie Zelenski, has become a classic in CS106B and X. Your mission is to write a program to play the game of Boggle™, which should help you get over any lingering doubts about the power of recursive techniques.

### 1. What is Boggle?

The game of Boggle is played with a set of sixteen letter cubes, which are standard six-sided dice except that they are marked with letters of the alphabet instead of numbers. The cubes are rolled and arranged into a 4x4 square that might look like this:

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

The object of the game is to start at one cube and then work through a chain of letters to form a word that meets the following conditions:


- The word must be at least four letters long.
- The path traced out by the letters in the word must be connected horizontally, vertically, or diagonally. You can't skip over intervening cubes to get the next letter.
- Each cube may be used only once in a given word.

For example, the sample pattern contains the word **PEACE** as follows:

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

The pattern, however, does not contain the word **PLACE**, which would require jumping from the **P** to the **L** and then back to the **A**. Similarly, it is not possible to make the word **POPE**, because doing so would require reusing the **P**.

Because the computer has a complete dictionary and you do not, we've tried to make the game a little more interesting by letting you find as many words as you can and then turning the computer loose to find the rest. If you're thorough, you can beat the computer because it is not allowed to count words you've already found. Most games, however, still end up as a rout, as illustrated by the following diagram of the Boggle display at the end of a game:

<b>Me</b>	<b>9</b>		<b>Computer</b>	<b>56</b>
lean peel clean pace pent lent bent clan				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

The numbers on the top line represent the scores for the human and computer players, respectively. Four-letter words score 1 point, five-letter words score 2, and so forth, with each additional letter contributing one additional point. The human player in this example has found eight four-letter words and one five-letter word for a total of 9 points.

## 2. Overview of the requirements

Your job in this assignment is to implement the Boggle game, which requires extensive use of recursive backtracking. Because we want you to concentrate on the strategy section, we have given you the following two modules to start with:

1. The `gboggle.h` interface implemented by `gboggle.c`, which is responsible for maintaining the graphical display.
2. The `lexicon.h` interface implemented by `lexicon.lib`, which is responsible for keeping track of a list of legal words.

The interfaces for these modules appear in section 3.

Even though these modules are given to you, the remaining part of the Boggle code contains most of the interesting programming. Your code, for example, must accomplish the following tasks:

- Set up the faces of the various letter cubes by reading data from a file. For each game, your program will need to shuffle the letter cubes, pick a random side from each one, and then call functions in `gboggle.h` to display the starting configuration on the screen.
- Implement the user's turn by accepting a list of words from the user. For each word, the program must check that it is legal and, if so, use the `gboggle.h` facilities to record the words that are found, highlight them on the screen, and keep track of the score.

- Implement the computer's turn by performing an exhaustive search to find all the acceptable words that the human player has not already found.
- Keep track of various housekeeping matters such as initializing the game, providing instructions, offering the user a chance to play again, and so forth.

Each of these operations is described in more detail in the sections that follow.

### Setting up the letter cubes

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 make sure that the computer version of the game works the same way, the starter folder includes a file `cubes.dat`, which contains the following data:

```
LRYTTE
VTHRWE
EGHWNE
SEOTIS
ANAEEG
IDSYTT
OATTOW
MTOICU
AFPKFS
XLDERI
HCPOAS
ENSIEU
YLDEVR
ZNRNHL
NMIQHU
OBBAOJ
```

Each of the 16 lines in the file consists of a six-character string in which the characters indicate what letters appear on the six faces of each cube. During the initialization phase, your job is to read this data file into a suitable data structure. For each game, your program must shuffle the letter cubes into a new configuration and pick a random side of each cube to display on the screen. Picking a random side is easy; the hard part is shuffling the cubes into a new configuration. The best way to shuffle is to mimic the selection sort algorithm (see p. 285 in *Programming Abstractions in C*), changing the outer loop so that each cycle chooses a random element instead of using an inner loop to find the smallest. This strategy is described in more detail in exercise 12-9 on page 450 of *The Art and Science of C*, which states the following:

Many algorithmic problems are related to sorting in their solution structure. For example, you can shuffle an array by "sorting" it according to a random key value. One way to do this is to begin with the selection sort algorithm and then replace the step that finds the position of the smallest value with one that selects a random position. The result is a shuffling algorithm in which each possible output configuration is equally likely.

### The user's turn

During the human player's turn, 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
- That it is defined in the lexicon as a legal word
- That it can be legally formed from the words on the board
- 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 the word to the user's word list and then update the score appropriately. In addition, you should use the facilities provided by the `gboggle.h` interface to highlight the word. More precisely, 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.

### 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 by the player.

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 detect as soon as possible that no words are possible down a particular path. For example, no words in English begin with the letters **ZX**. Thus, if you are searching through the board and have built a path starting with this letter combination, you can stop right there because no additional letters will end up making a word. Note that the `lexicon.h` interface exports an `IsPrefix` function that allows you to determine whether a given set of letters ever appears at the beginning of a word.

## 3. The supplied modules

The **Assignment #6** materials on the web, when expanded, contain the following files:

<code>gboggle.h</code>	The interface to the library of graphics routines for Boggle
<code>gboggle.c</code>	The implementation of <code>gboggle.h</code> , presented in source form
<code>lexicon.h</code>	The interface to the lexicon abstraction used to maintain word lists
<code>lexicon.lib</code>	A precompiled implementation of <code>lexicon.h</code>
<code>cubes.dat</code>	A data file indicating the faces of the Boggle cubes
<code>ospd2.dat</code>	A lexicon data file from the game of Scrabble™.
<code>boggle.mcp.rsrc</code>	A Mac resource file for adding sound (these are separate <code>.wav</code> sound files for PCs)
<code>Boggle Demo</code>	A Mac executable version of the Boggle game, or
<code>Boggle.exe</code>	A PC executable demo

When you create your starter project, you will need to add `gboggle.c` and `lexicon.lib` to it, as well as your own source code file, which you should call `boggle.c`. Also add `boggle.mcp.rsrc` if you are using sound on a Mac.

### The `gboggle.h` interface

The `gboggle.h` interface is used to manage the appearance of the Boggle 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. 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. A complete listing of the interface appears in Figure 1.

**Figure 1. The `gboggle.h` interface**

```

/*
 * File: gboggle.h
 * -----
 * The boardGraphics.h file defines the interface for a set of
 * routines that draw the boggle board and manage the word lists and
 * scoreboard graphics.
 */

#ifndef _gboggle_h
#define _gboggle_h

typedef enum {Human, Computer} Player;    //identifies players

/* 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 using
 * InitGraphics() to erase the graphics window. It will re-draw 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('D', 0, 3);
 * -----
 * 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), lower right is (3,3). The above usage would put a D in
 * the top right corner cube.
 */
void LabelCube(char ch, int row, int col);

```

```

/* Function: HighlightCube
 * Usage: HighlightCube(2, 2, TRUE);
 * -----
 * This function highlights or unhighlights the cube specified by
 * position (row, col). It is intended to be used to show which
 * letter are used to form a given word on the boggle board. The
 * highlighting will invert the specified cube to draw attention to
 * it. The "lit" parameter determines whether you are drawing the
 * darkened cube or erasing it. TRUE indicates the dark cube should
 * be drawn, FALSE indicates the dark cube should be erased.
 */
void HighlightCube(int row, int col, bool lit);

/* Function: RecordWordForPlayer
 * Usage: RecordWordForPlayer("fruitfly", Human);
 * -----
 * This function draws the new word added to the player's word list
 * display and updates the scoreboard accordingly. The two players
 * are identified by the enumerated type Player (see tyedef above)
 * Scoring is calculated this way: a 4-letter word is worth 1 point,
 * 5-letter is worth 2 points, and so on.
 */
void RecordWordForPlayer(string word, Player playerNum);

#endif

```

### The `lexicon.h` interface

Although the word *lexicon* is often used as a synonym for *dictionary*, it doesn't imply that the words have definitions in the way that dictionary entries do. We have used the word *lexicon* for this abstraction to differentiate it from the symbol table that appears later in the text. A symbol table associates a key with a definition; in a lexicon, the definitions are superfluous. All you need to know is whether a word is in the list. The lexicon abstraction provides this capability.

The contents of the `lexicon.h` interface are shown in Figure 2 on the next page. 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, which turns out to be very useful in designing the computer's strategy.

The lexicon is an example of an *abstract data type*, which is often abbreviated to *ADT*. As you will learn later in the course, the behavior of an ADT is defined by a set of functions exported by an interface, but its implementation is kept hidden from the client. For this assignment, the implementation of the lexicon abstraction is given to you as a precompiled library called `lexicon.lib`. You don't need to know how it works to use it. All you need to know is that you can use the functions in the interface to create new lexicons, add words to them, check if words are defined, and so forth.

Figure 2. The lexicon.h interface

```

/* File: lexicon.h
 * -----
 * Defines a lexicon abstraction, or word list. You create an empty
 * lexico and then are able to add individual words or read words from
 * a file of words in compressed binary representation. You can then
 * determine whether a word is valid lexicon entry and whether a
 * prefix is valid for any word in that lexicon.
 */

#ifndef _lexicon_h
#define _lexicon_h

#include "genlib.h"

/* Type: lexiconADT
 * -----
 * Provides an abstract data type for manipulating a lexicon. The
 * type is a purely abstract type in this interface, defined entirely
 * in terms of the operations. The client has no access to the record
 * structure used to implement the actual type.
 */

typedef struct lexiconCDT *lexiconADT;

/* Operations */

/* 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 all 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,"ospd2.dat");
 * -----
 * 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: count = WordsInLexicon(lexicon);
 * -----
 * Returns the number of words in the lexicon.
 */
long WordsInLexicon(lexiconADT lexicon);

/* Function: IsWord
 * Usage: foundIt = IsWord(lexicon, "dork");
 * -----
 * This function looks up the given word in the lexicon and returns a
 * boolean result on whether this word is a valid entry.
 */
bool IsWord(lexiconADT lexicon, string word);

/* Function: IsPrefix
 * Usage: isGood = IsPrefix(lexicon,"rgs");
 * -----
 * This function reports whether any words at all in the lexicon begin
 * with the specified prefix. A word is defined to be a prefix of
 * itself. The empty string is a prefix of everything.
 */
bool IsPrefix(lexiconADT lexicon, string prefix);

#endif

```

## 4. Planning your implementation

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. When Julie Zelenski first assigned Boggle two years ago, she recommended that students break the problem down into the following five phases:

- *Phase 1—Generate and display the initial board.* In this part of the problem, you need to read in the data file for the letter cubes, shuffle the cubes, and then call functions in the `gboggle.h` interface to display the starting configuration on the display.
- *Phase 2—Accept words from the user and check all conditions except for whether the word exists on the board.* In this phase, your program will need to read words from the user, make sure they meet the length requirement and are in the dictionary, and then display the word on the screen if it passes the conditions.
- *Phase 3—Find the user's word on the board and highlight it.* After completing this phase, your program should be able to act as a referee for the player's turn.
- *Phase 4—Add the code for the computer's turn.* This phase is where the most difficult applications of recursion come into play. It is easy to get lost in the recursive decomposition here and you should think carefully about how to proceed.
- *Phase 5—Complete all the bookkeeping aspects of the game.* In this phase, you should add in the instructions, make sure that the user is given a chance to play a new game, and all of the other bits and pieces required to finish off the Boggle application.

## 5. Possibilities for extension

As with most assignments, Boggle admits 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. Implementing **Qu** as a single unit requires several revisions to the current code.
- *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 appropriate letter cubes.
- *Reduce the level of memory utilization.* Implementations that use the `strlib.h` library are likely to be wasteful of memory because functions like `Concat` dynamically allocate new memory each time they are called. If you don't free this memory, the program will eventually run out of memory. Try to solve this problem either by freeing the memory you no longer need or by avoiding the dynamic allocation in the first place (the latter is the easier approach).
- *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.
- *Add sound to your application.* You will be given no extra credit for adding sound (it's very simple), but it can be kind of fun. It is very easy to play prerecorded sounds from a resource file (or `.wav` files on a PC). If you are using CodeWarrior, add the `boggle.π.rsrc` file to your project, (just like your `.c` files). If you are using Visual C++, make sure all the `.wav` files for the sounds are in the same folder as your project. Then (in either environment), add `#include "sound.h"` to your code, and use the function `PlayNamedSound` from `sound.h` to play sounds. For example, if you have a sound called "Whoops", you can play it using the function call `PlayNamedSound("Whoops")`.

The resource file called `boggle.π.rsrc` that contains the following named sounds:

```
"come on faster"
"not"
"That's Pathetic"    (Note: there is no apostrophe in the name of this file on the PC)
"Excellent"
"Denied"
"Dice Rattle"
"yeah right"
"whoops"
"not fooling anyone"
"oh really"
"Moo"
"Tinkerbell"
"tweetle"
"Idiot"
"yah as if"
```

The PC implementation contains most of these same sounds, but as separate `.wav` files. You do not need to add these to your project. Note that on the PC there is no apostrophe in "Thats pathetic".

## 6. Deliverables

Submit any `.c` and `.h` files that you write. Most likely, this will be the single file `boggle.c`.