ECE 209
Fall 2020

Program 2: `boggle`

Due Friday, Oct 2 @ 11:59pm

This programming assignment must be completed individually. Do not share your code with or receive code from any other student. Evidence of copying or other unauthorized collaboration will be investigated as a potential academic integrity violation. **The minimum penalty for cheating on a programming assignment is a grade of -100 on the assignment.** If you are tempted to copy because you're running late, or don't know what you're doing, you will be better off missing the assignment and taking a zero. Providing your code to someone is cheating, just as much as copying someone else's work.

DO NOT copy code from the Internet, or use programs found online or in textbooks as a "starting point" for your code. Your job is to design and write this program from scratch, on your own. Evidence of using external code from any source will be investigated as a potential academic integrity violation.

This program implements part of a computer-aided version of the word game, Boggle.

The learning objectives of the program are:

- Write C functions according to a specification.
- Use pseudorandom numbers to implement dice rolling and array permutations.
- Write functions with string and array parameters.

- Write a function that searches for a specified pattern of data.

WARNING: This game involves using random letters to spell words. It's possible, even likely, that some offensive or inappropriate words may appear. We must maintain a respectful, professional, and inclusive environment in the class. Do not make lewd comments or jokes about words in the game with your classmates or friends. Reports of such behavior may be treated as harassment (section 10.11 of the Code of Student Conduct).

Background

Boggle is a word game that was published by Parker Brothers in the 1970s, and is currently sold by Hasbro. The player's task is to find as many words as possible in a random 4x4 grid of letters. This program will be an approximation of the Boggle game: we'll call it Boggle-ish. We'll model various components of the game, described below, but we'll avoid some of the harder parts, like validating that a string is actually a word in the dictionary.

The most challenging part of our program will be to verify that a given string can be found in a 4x4 grid of letters.

In computer games, we often need to model random events, such as rolling dice or shuffling cards. We do this by generating random numbers, and then using those numbers to represent the physical world. For example, a random number between 1 and 6 can represent a roll of a single six-faced die.

Generating truly random numbers in a deterministic computer is considered a hard problem, and requires some hardware help. (By the way, quantum computers are good at generating truly random numbers, but that's a topic for another day.) Instead, computers generate pseudorandom numbers by mathematically creating a sequence of numbers that "look random." The numbers are easy to generate, but hard to predict unless you know the formula and the initial value in the sequence.

There are some functions in the standard C library for generating pseudorandom numbers, as well as code available online. For this assignment, code is provide for one particular method. We use this code, rather than the standard library functions, to make sure that the same sequence of numbers will be generated on any platform (CLion, ZyBook, etc.). This will make testing possible.

Because it's tiresome to keep writing (and reading) "pseudorandom," I will often use the term "random" in the text below.

Rules of Boggle-ish

This program provides allows one or more humans to play Boggle with the computer, rather than using a physical Boggle game. Here's how the program will work:

1. Player enters a ***seed*** number that will determine the sequence of random numbers used throughout the run. This allows the same game to be played over and over, which will aid in debugging and testing.
2. Player enters a ***time limit***, in seconds. This is the amount of time the program will pause to allow humans to search for words in the grid of letters. (This may be zero, if the player wants to skip the timer.)
3. The program then "shakes" the virtual tray of letters and prints the 4x4 grid.
4. The program will pause for the number of seconds specified, printing cues as time progresses. During this time, the human(s) should write down (on paper) as many words as possible, following the rules below.
5. When the timer expires, the program prompts the player to ***enter a word*** that was found. The program will (a) verify that the word can actually be found in the grid, and (b) assign a score based on the length of the word.
6. Step 5 repeats, and the total score is accumulated, as long as the player keeps typing words. The player enters "q" to quit, and the total score is printed.

What are the limitations of Boggle-ish? First, we can't enforce the time limit. Once timer expires, the player can take as much time as she wants to enter as many words as she wants. Second, we won't confirm that the word is actually a word; it could be any string of letters that can be found in the grid. Third, we won't detect if the player enters the same word multiple times.

	0	1	2	3
0	E	O	Qu	P
1	G	R	I	A
2	G	L	S	N
3	A	L	G	O

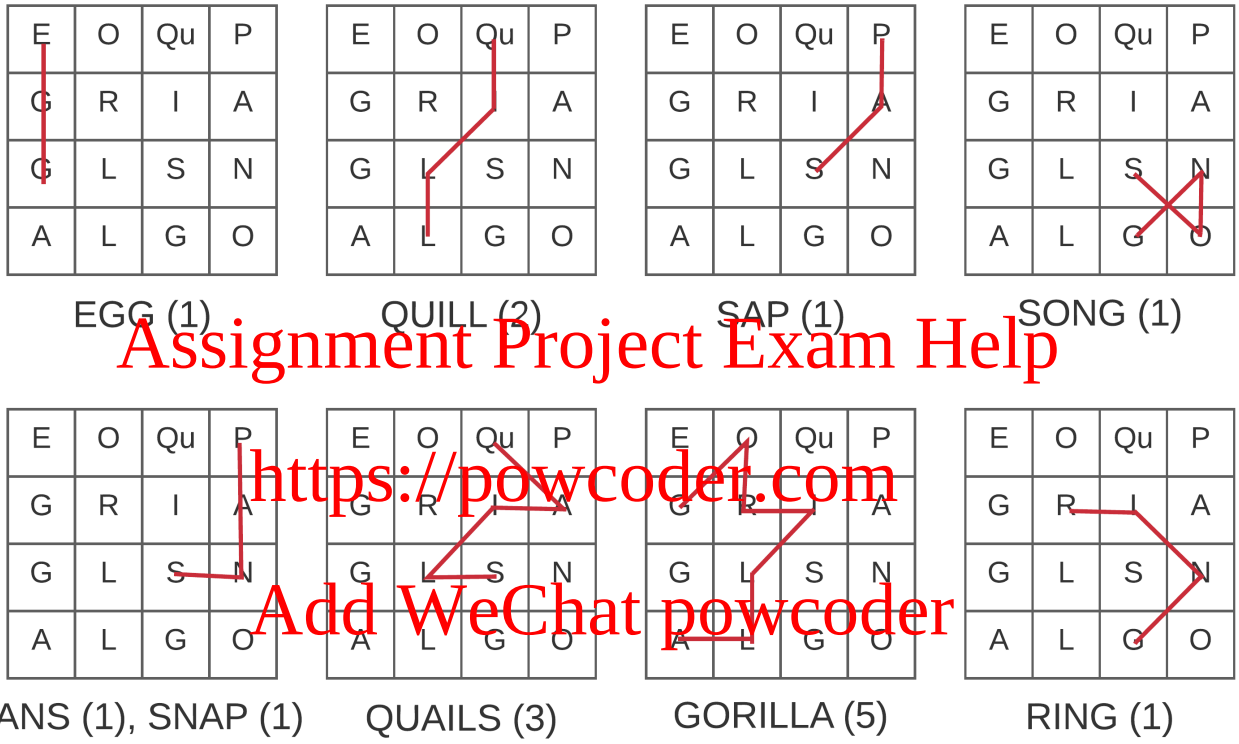
In the program specification below, we will discuss our approaches for organizing data and functions to carry out these steps, but first, let's discuss the rules for finding/validating words in the grid of characters. The figure to the left shows a sample 4x4 grid of letters.

Our job is to find words using these letters, in this arrangement, with the following rules:

- A word is three letters or more.

- A word is composed of a *sequence* of letters in adjacent blocks. Adjacent can be up, down, right, left, or diagonal in any direction.
- There is no "wrap around." Words may not pass the boundaries of the grid.
- A word cannot use the same block more than once.
- The letter 'Q' automatically includes a 'U' after it, and it counts as two letters in the word. (You can't create words in which 'Q' is not followed by 'U'.)

The figure below shows several words found in our sample grid, along with their scores. Scores are based only on the word length, as described later.



There are many other words available in this grid. See if you can find SNAIL, GORE, GRILLS. You can use both RING and RINGS. You can do SNAIL and NAILS, but not SNAILS, because that would use the S block twice in the same word.

The score is based on the length of the word, with "Qu" counting as two letters. Scores are given in the table to the right.

Length	Points
3, 4	1
5	2
6	3
7	5
8+	11

Program Specification

You are given a **main.c** program to get started. You must use this template -- don't write your own. The program will be in one file. Your job will be to implement specific functions needed by this program. The required functions are declared at the top of the file. Write the definitions of these functions (and any others you create) *below* the **main** function.

There are some global variables (what???) declared and initialized in the template. You must use these variables. Do not add more global variables to the program.

There are two functions already defined for you. The **lfsrNext** function implements the linear feedback shift register described below for generating pseudorandom numbers. The **startTimer** function uses the standard library sleep function to implement the timer part of the program. (NOTE: The timer will not be used in the ZyBook tests. It's provided to make the program more interesting, but you can always set the time to 0 while you're debugging.)

The functions that you must implement are:

```
void seedRandom(unsigned int seed);
   // initialize the PRNG if seed is not zero
unsigned int getRandom(unsigned int limit);
   // return a pseudorandom number between 0 and limit-1

void permute(unsigned int values[], size_t n);
   // create a permutation of an array of size n

void shakeTray();
   // randomize the letter grid
void printTray();
   // print the letters in the grid
bool findWord(const char *word);
   // determine whether the word can be found in the grid
unsigned int scoreWord(const char *word);
   // return the score of a word
   // returns zero if word is too small or not in the grid
```


Implementation

Pseudorandom numbers

The technique we will use for generating random number is called a linear feedback shift register (LFSR). The basic idea is that certain bits of an n-bit unsigned integer are processed in some way, then the value is shifted and the new bits are inserted. If you do this correctly, you end up with a sequence that cycles through all (or most) of the non-zero n-bit values in a pseudorandom order. Refer to this Wikipedia article for more information.

We are using a global variable to hold the LFSR state. I've told you to only use a global variable if there's a very good reason, so what is my justification? (1) We need to share this information among mutiple functions. Doing this through parameter passing (the preferred approach) complicates the interface and makes the capability harder to use. (2) We only need one sequence of LFSR values, so we only need one variable to hold the state, rather than keeping track of multiple local copies.

We can come up with different approaches to solve this problem that don't require the use of a global variable[1], but this is simple and justified. We don't want this global variable to used anywhere outside of the random number functions we have declared. In other words, the rest of the program will pretend like the global variable is not there. Your code must only call **getRandom** and **seedRandom** -- you may not access the global variable directly. (Except when you are defining those two functions, of course.)

[1] We could use a static local variable.

To seed the pseudorandom number generator (PRNG), just write the seed value into the global variable. You must, however, protect against setting the value to zero, because that will just result in an endless sequence of zeros. (Not very random!)

The LFSR we're using is 16 bits. Therefore, if you read the value itself, you will get a number between 1 and 0xFFFF. (Important: Zero is not a legal value!) Calling **lfsrNext** will change the global variable to the next value in the sequence.

The **getRandom** function must advance the LFSR to its next state (using **lfsrNext**), and then convert that value to a number between 0 and (limit-1). Hint: The modulo operator will serve an important role. It is important that you only call **lfsrNext** *once* for each call to **getRandom**. Otherwise, you will get a different sequence of random numbers, and the tests will fail.

Reminder: You must not directly access the global variable from any of your other functions.

Data Structure: Tray and Dice

An important part of solving a problem using computers is to determine the data that needs to be manipulated, and how that data should be represented. In this case, I've made this decision for you, but in the future, you will want to think about this aspect of your solution. The right choice of data representation can make the problem easier to solve, and the wrong choice can make it harder to solve.

For this program, we will again use global variables (gasp!) to hold the information about the Boggle dice and the tray that determines their arrangement in the grid. The justification for global variables is very similar to the case outlined above: (1) Information is needed by multiple functions. (2) Passing the information makes the function interface more complicated, and gets in the way. (3) There is no need for this program to have multiple instances of the tray, so there's no need to keep separate local copies.

First, let's talk about the dice. The Boggle game comes with a set of 16 dice. Each die has a set of letters on its faces, and those letters are different for each die. A single die can be represented as an array of six letters, one per face.

We have a collection of dice, so can we have an array of arrays? Yes, in fact, we can. This is called a two-dimensional array, and it uses two indices instead of one. For example, consider this 2D array of integers:

```
int data[5][10];
```

This can be interpreted as a 5-element array, where each element is a 10-element array. The total number integers is 50. You can think of this as a 2D matrix, with 5 rows and 10 columns. Element `data[i][j]` is the integer in row *i* and column *j*.

In our case, we want an array of 16 die, and each die is an array of 6 letters:

```
const char dice[16][6];
```

In this case, the indices correspond to the die number and the face on that die. In otherwords, the first face on die #3 is dice[3][0]. Note that I've declared this as const, meaning that the array data may be read but not written. (You can't change the number of dice or the letters on the dice without recompiling the program.)

I have used initializers to set the values of the dice, as follows:

```
const char dice[16][6] = {{'A', 'A', 'C', 'I', 'O', 'T'}, // die 0
                          {'A', 'H', 'M', 'O', 'R', 'S'}, // die 1
                          {'E', 'G', 'K', 'L', 'U', 'Y'}, // die 2
                          ...
```

The initializer for a multi-dimensional array uses a nest braces notation. See ZyBook 6.11 for more examples. Note that each die has different letters. This data is taken from the traditional Boggle set sold in the US in the 1970s. There are other variants for different languages, and the official set of letters has changed over the years.

NOTE: The letter 'Q' appears on die 10. Since these are characters, we can't put "Qu" (a string), and 'Qu' doesn't make sense. So your code needs to "know" that 'Q' means "Qu" when you are printing and using this letter.

Now we need to represent the grid, or the tray in the Boogle game. You might think that we'd use a 2D array, since it has four rows and four columns. However, manipulation of this data is easier if we use a regular 1D array and map the indices to the 2D grid. (Trust me on this.)

There are two global variables used to represent the grid:

```
unsigned int trayDice[16];
char trayLetters[16];
```

In both cases, each array element represents one position in the 4x4 grid. The mapping of index to row and column is given by the following formula:

$$\text{index} = (\text{row} \times 4) + \text{column}$$

In the grid, row 0 is the top row, and column 0 is the leftmost column. The rows and columns are labeled in the figure on page 2.

The `trayDice` array represents the die which occupies each block of the grid. When we "shake" the tray, we move these numbers around to put each die in a new position.

The `trayLetters` array represents the letter that is face up for each position. This must be one of the letters associated with the die in that position.

Example: If `trayDice[0]` is 4, that means die 4 is in the upper-left position of the grid. One of that die's six letters will be face up, which means `trayLetters[0]` must be one of A, C, D, E, M, or P, corresponding to `dice[4][x]` for some value x.

Permutation

When we take an array and rearrange the values without changing any value, that is called a permutation. Since our die numbers will always be 0, 1, 2, ..., 15, then rearranging the dice in the tray is a permutation of the `trayDice` array.

The **permute** function must be written to perform this permutation. While there are multiple ways to perform a permutation, your definition must use the algorithm described below, so that every program with the same random seed will generate the same permutation. This is important for testing.

The permute function must be defined to work with an array of any size. The caller will pass in the array and the size -- do NOT use the global `trayDice` array for this function.

We will use the Fisher-Yates shuffle algorithm to perform the permutation of an array v of size n. You must implement using this algorithm, as described here. Do not copy code from the internet. Write the code yourself.

Given array **v** of size **n**:

For each **i** from **n**-1 to 1,
    Choose a random index **j**, where $0 \leq j \leq i$.
    Swap **v[i]** with **v[j]**.

Note that j could be the same as i. This is important, because it's OK for an array element to stay in the same place.

Given the same starting array and the same random seed, your **permute** function should always generate the same permutation. More importantly, it must also generate the same permutation as anyone else's implementation, which allows us to automate the testing.

Shaking the tray

In the Boggle game, shaking the tray has two effects. (1) It changes the position of the die in the tray. (2) It changes which letter is facing up on each day. Given the functions you've implemented thus far, here's how to model this action in the program:

(1) Generate a permutation of the `trayDice` array.

(2) For each position i, `trayDice[i]` tells which die is in that position. Select a random face of that die to be up, and record the letter in `trayLetters[i]`.

Printing the tray

To print the tray, print the four letters in the top row, followed by the four letters in the next row, etc. To simplify things, print the letter followed by two spaces, even if it's at the end of a row. However, if the letter is 'Q', print "Qu" followed by one space.

With a monospaced font, this will display a nice, neat 4x4 grid of letters on the console.

Scoring a word

Once the player starts entering words, your **scoreWord** function will be called for each word. If the word is less than three characters, it will have a score of zero. If the word is more than 16 characters, it couldn't possibly be found on the grid, so it will also have a score of zero.

For any other word, call **findWord** to see if it is found on the grid. If not, the score is zero. If so, return a score based on the table shown earlier.

Hint: This is a fairly easy function to write, while **findWord** is much more challenging. My recommendation is to start with a **findWord** function that always returns true, so that you can concentrate on just getting this function working properly. Once you're done, move on to an actual implementation of **findWord**.

Finding a word

The **findWord** function is the most challenging in this program. Given a string, you must determine whether any legal sequence of positions in the grid can be used to create the word. Your function simply returns true (1) if yes, false (0) if no.

The string you are given will only contain upper-case letters, so you don't have to worry about checking for bad characters or for lower-case letters.

The basic idea is reasonable. Look for a block that contains the first letter of the word. Then look at the neighbors of that block for the next letter. (An alternate approach: Find any occurrence of the second letter, and see if that block is a neighbor of the first.) If there's no match, move on to see if there's another occurrence of the first letter, and try again.

If there is a match of the second letter, in a neighboring block, then you need to search its neighbors for the third letter, and so forth.

The challenging part will happen when you match some letters, but then fail. You can't just declare defeat, because there might be other combinations that will work. You will need to "backtrack" and keep searching, but you also must remember this failure, so that you don't find it again.

There are a number of ways to attack this problem, and I want you to use your problem-solving skills. Try it yourself, before asking for help. You'll learn more that way, even if you don't come up with a complete solution on your own.

One suggestion is to use a recursive algorithm. (ZyBook Chapter 10, and maybe you talked about recursion in ECE 109?) We can define a function that asks: Does string X appear in the grid, starting at position Y? (This is not the **findWord** function, because there is no starting position passed to that function.)

Let's say we're looking for the string "ABCD". We find 'A' at position 3. We call our function that asks: Does "ABCD" appear at position 3? The answer is yes if (a) we find 'A' at position 3, (b) we find 'B' at one of the neighboring positions (let's call that position N), and (c) the string "BCD" appears at starting position N.

See the recursion? We can do that last part by calling the function itself, with a new string and a new starting position. Backtracking is handled by the calling stack; if the call in part (c) returns false, we can try a different neighboring position. If we run out of neighboring positions, we return false.

You also have to pass along some other information, to make sure that you don't use the same block twice in the same word.

This code is, in my opinion, much easier to write using recursion, but there are other alternatives. You can use a loop to look for each letter, and record the sequence of positions in an array, so that you don't visit the same block twice. You can use an array to record the previous positions, because you know that the longest matching string will be 16 characters or less.

Developing and Submitting the Program

You will submit this code as a zyBook lab, but you will develop the program outside of the zyBook. It is expected that you will use CLion, but you are free to use whatever development environment that you want.

1. In CLion, create a new project. Specify a C Executable, and use the C11 standard. Name the project whatever you like, but I recommend something meaningful like "prog1" or "cipher". This will create a directory in the place where your CLion projects are kept. Exit CLion.
2. Go to Chapter 13 in the zyBook, and look for a lab titled: *Fall 2020, Program 2: boggle*.
3. Download the **main.c** file provided for you, and overwrite the **main.c** file in the CLion project that you created in Step 1.
4. Now use CLion to complete the program, using the editor, compiler, and debugger to meet the program specifications.
5. When you are ready, upload your **main.c** file to the zyBook assignment and submit. This will run the tests.

If some tests fail, go back to Step 4 and work on your program some more. The input for the failed tests will give you some idea of what's not correct, but use the debugger to figure out what's happening with your implementation.

Several iterations of Steps 4 and 5 might be necessary. In fact, it's a reasonable strategy to write a program that only passes the first test (or some tests), then improve it to pass the next test, etc. (This even has a fancy name: test-driven development.) Write dummy versions of all the functions, so that the code compiles. For example, write a version of scoreWords that always returns 0, a version of findWord that always returns false, a version of permute that doesn't change anything, etc. As you move to different features, you will have code that compiles, and you can focus just on the part you're trying to solve.

There is no limit to the number of times you can submit. Each submission will overwrite the earlier ones, so make sure that your new code does not break tests that passed earlier.

Hints and Suggestions

- Don't overcomplicate the program. Do not use anything that we have not covered in class. (For example, don't try to use pointers or string library functions.)
- Work incrementally. Get one part of the code working, and then move to the next.
- Use a timer value of zero during development and debugging.
- For compiler errors, look at the source code statement mentioned in the error. Try to figure out what the error is telling you. Try to fix the first error first, and then recompile. Sometimes, fixing the first error will make all the other errors go away. (Because the compiler got confused after the first error.)
- Use a source-level debugger to step through the program if the program behavior is not correct. If you are using CLion on your own computer, the debugger is integrated with the editor and compiler, so there's no excuse for not using it.
- For general questions or clarifications, use Piazza, so that other students can see your question and the answer. For code-specific questions, post a private message to Piazza and attach your code as a file. (Do not copy and paste!)

Administrative Info

Updates or clarifications on Piazza:

Any corrections or clarifications to this program spec will be posted on Piazza. It is important that you read these postings, so that your program will match the updated specification.

What to turn in:

- Submit your **main.c** file to the zyLab assignment in 13.2, "Fall 2020, Program 2: boggle".

Grading criteria:

20 points: Program submitted on time. Compiles with no warnings or errors. As long as you've made some effort, and do not just submit the template file, you will get these points.

10 points: Proper coding style, comments, and headers. No global variables. No goto. See the Programming Assignments section on Moodle for more style guidelines. (You will not get these points if you only submit trivial code.)

10 points: Correct implementation of **seedRandom** and **getRandom**.

15 points: Correct implementation of **permute**. (Relies on correct random functions.)

10 points: Correct implementation of **printTray**.

10 points: Correct implementation of **shakeTray**. (Relies on correct random functions and correct permute.)

10 points: Correct implementation of **scoreWord**. (If **findWord** does not pass, will grade manually.)

15 points: Correct implementation of **findWord**.

NOTE: The ZyBook tests will only total 70 points. The first 30 points will be assigned manually by the grader.

NOTE: Points may be deducted for errors, even if all of the zyBook tests pass. This will be rare, but it may happen if it is obvious to the grader that the program is written specifically to pass the specific tests, and would not pass other similar tests.