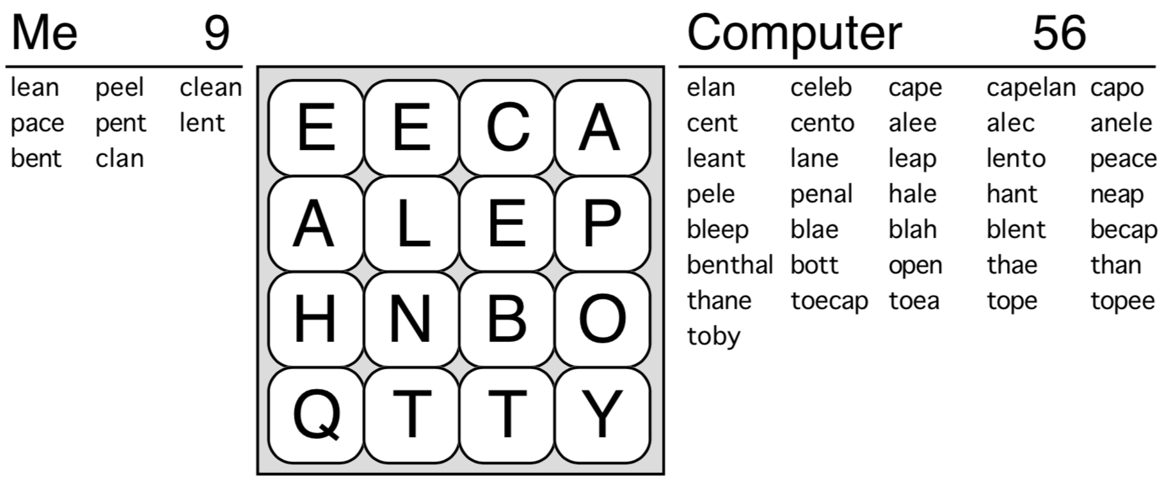
Lab 6: Boggle

Deadline: 2019-10-10 10:10:10 pm.
Please compress your source code as `lab6-STUDENTID.zip` and upload it to ftp://dmkaplony:public@public.sjtu.edu.cn:/upload/c++2019/lab6/ before the deadline.

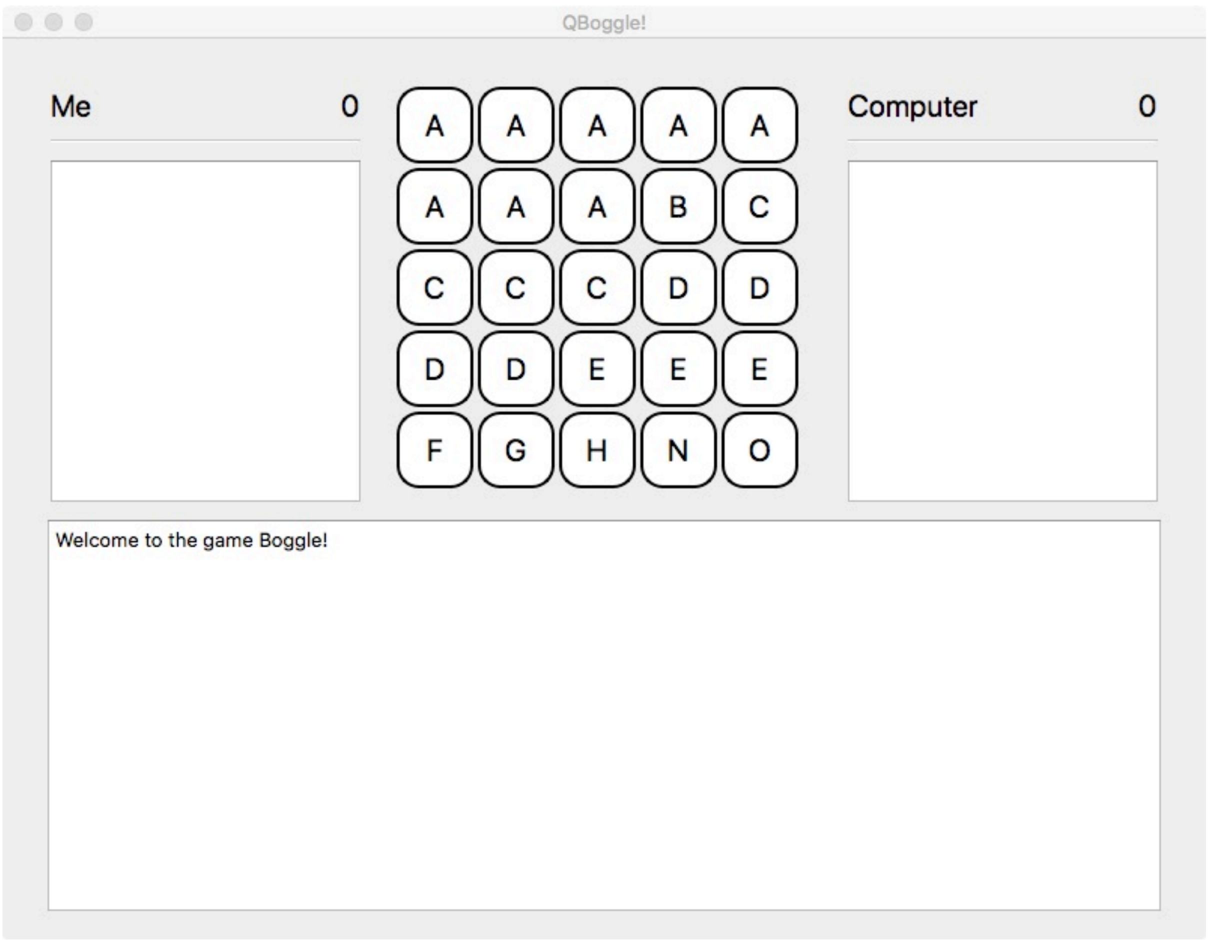


Me 9

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				



Welcome to the game Boggle!

The game of Boggle

The Boggle game board is a square grid onto which you randomly distribute a set of letter cubes. The goal is to find words on the board by tracing 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. Each cube can be used at most once in a word. In the original version, all players work concurrently listing the words they find. When time is called, duplicates are removed from the lists and the players receive points for their remaining words.

Your assignment is to write a program that plays a fun, graphical rendition of this little charmer, adapted for the human and computer to play pitted against one another. You'll quickly learn you have little hope of defeating the computer, but it's awesome to realize you wrote this program that can so soundly thrash you again and again!

This assignment will give you a little more practice as a class client, but the main focus is on designing and implementing recursive algorithms. At the heart of the program are two recursive functions that find words on the board, one for the human player and another for the computer. Although each function is only a dozen lines of code, recursive algorithms can be dense and tricky and will require your best recursion mojo to conquer. When you are done, you will have earned the right to call yourself a journeyman in the way of recursion.

How's this going to work?

You set up the letter cubes, shake them up, and lay them out on the board. The human player gets to go first (nothing like trying to give yourself the advantage). The player enters words one by one. After verifying that a word is legitimate, you highlight its letters on the board, add it to the player's word list, and award the player points according to the word's length.

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 mercilessly, but the player is free to try again and again, until finally ready to acknowledge the superiority of silicon.

The letter cubes

The letters in Boggle are not simply chosen at random. Instead, the letter cubes are designed 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 an array of the cubes from the original Boggle. Each cube is described using a string of 6 letters, as shown below:

```
const string BIG_BOGGLE_CUBES[25]  = {
   "AAAFRS", "AAEEEE", "AAFIRS", "ADENNN", "AEEEEM",
   "AEEGMU", "AEGMNN", "AFIRSY", "BJKQXZ", "CCNSTW",
   "CEIILT", "CEILPT", "CEIPST", "DDLNOR", "DDHNOT",
   "DHHLOR", "DHLNOR", "EIIITT", "EMOTTT", "ENSSSU",
   "FIPRSY", "GORRVW", "HIPRRY", "NOOTUW", "OOOTTU"
};
```

These strings are used to initialize the cubes on the board. At the beginning of each game, "shake" the board cubes. There are two different random aspects to consider. First, the cubes themselves need to be shuffled so that the same cube is not always in the same location on the board. Second, a random side from each cube needs to be chosen to be the face-up letter.

To rearrange the cubes on the board, you should use the following shuffling algorithm, presented here in pseudocode form:

```
Copy the constant array into a vector vec so you can modify it.
Shuffle vec using the following approach:
  for (int i = 0; i < vec.size(); i++) {
    Choose a random index r between i and the last element position, inclusive.
    Swap the element at positions i and r.
  }
Fill the Boggle grid by choosing the elements of vec in order.
```

This code makes sure that the cubes are arranged randomly in the grid. Choosing a random side to put face-up is straightforward. Put these two together and you can shake the cubes into many different board combinations.

Alternatively, the user can choose to enter a custom board configuration. In this case, you still use your same board data structure. The only difference is where the letters come from. The user enters a string of characters, representing the cubes from left to right, top to bottom. Verify that this string is long enough to fill the board and re-prompt if it is too short. If it's too long, just ignore the ones you don't need. You do not need to verify that the entered characters are legal letters.

The human player's turn

The human player enters each word she finds on the board. For each word, check that:

- It is at least four letters long.
- It is contained in the English lexicon.
- It has not already been included in the player's word list (even if there is an alternate path on the board to form the same word, the word is counted at most once).
- It can be formed on the board (*i.e.*, it is composed of adjoining letters and each cube is used at most once).

If any of these conditions fail, the word is rejected. If all is good, you add the word to the player's word list and score. In addition, you graphically show the word's path by **temporarily highlighting** its cubes on the board. The length of the word determines the score: one point for a 4-letter word, two points for 5-letters, and so on.

The player enters a blank line when done finding words, which signals the end of the human's turn.

The computer's turn

The computer then searches the entire board to find the remaining words missed by the human player. The computer earns points for each word found that meets the requirements (minimum length, contained in English lexicon, not already found, and can be formed on board).

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. One of the most important Boggle strategies is to prune dead end searches. For example, if you have a path starting **zx**, the lexicon's `containsPrefix` member function will inform you that there are no English words down that path. Thus, you should stop right here and move on to more promising combinations. If you miss this optimization, you'll find yourself taking long coffee breaks while the computer is futilely looking for words like **zxgub**, **zxaep**, etc. Even though you may love coffee, this is obviously not the best idea.

Our provided resources

We provided an `EnglishWords.txt` file containing over 125,000 words, which is about four times as large as the average person's vocabulary, so it's no wonder that the computer player is so hard to beat!

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—Cube setup, board drawing, cube shaking.* Design your data structure for the cubes and board. It will help to group related data into sensible structures rather than pass a dozen parameters around. As usual, no global variables. Set up and shuffle the cubes. **You need to use a 5x5 board configuration**.
- *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 or that don't meet the minimum word length or that aren't in the lexicon. Add words to the graphical display and keep score.
- *Task 3—Find a given word on the board.* Now put your recursive talents to work in verifying that a word can be formed on the board, subject to the adjacency and non-duplication 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 move on to the next position. If a path is found, highlight the cubes that form the word.
- *Task 4—Find all the words on the board (computer's turn).* Now it's time to 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.
- *Task 5—Loop to play many games, add polish.* Once you can successfully play one game, it's a snap to play many. With everything now working, it's time to finish off the details. Be sure to gracefully handle all user input. Make sure your comments are thoughtful and complete. Pat yourself on the back and go eat some ice cream!
- *Task 6—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 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.

- *Task 7—Play by Mouse.* Since we have prepared a graphical user interface, the player could explore words by clicking on the board. Your program should track and highlight which cubes have been clicked (and thus, selected) and their order. Once the selected cubes form a valid word that hasn't been explored before, your program should award the player and append the word to the player's list. The selection and highlight should be cleared if 1) a word has been added to the player's list; or 2) the current selected cubes cannot form any word

Requirements and suggestions

Here are a few details about our expectations for your solutions:

- Words should be considered case-insensitively: **PEACE** is the same as **peace**.
- 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. They are somewhat similar, and you may be tempted to try to integrate the two into one combined function. In general, we applaud this instinct to unify similar code. However, we need to tell you that it doesn't play out well in this case. There are enough differences between the two that they don't combine cleanly and the unified code is actually made worse, not better, as a result. Given the tricky nature of recursion, you should focus on writing exceptionally clean code that clearly communicates its algorithm and thus can be easily maintained and debugged. An important goal for this assignment is that you to learn how to employ these two different varieties of recursion into the context of a larger program, and we expect you to do so by writing two separate recursive searches.
- Our solution is about 250 lines of code (not counting comments) and is decomposed into about 30 functions.

A little more challenge: for fun but not profit

Boggle has many opportunities for extension. You could consider the following extensions.

- *Board exploration*. As you will learn, some Boggle boards are a lot more fruitful that others. Write some code to discover things about the possible boards. Is there an arrangement of the standard cubes that produces a board containing no words? What about an arrangement that produces a longest word, maybe even using all the cubes? What is the highest-scoring board you can construct? Recursion will be handy in trying out all the possible arrangements, but there are a *lot* of options (do the math on all the permutations . . .), so you may need to come up with some heuristics to direct your explorations.