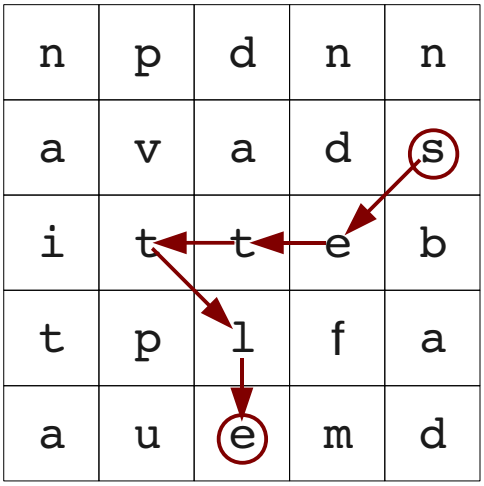
CS 215 - Fundamentals of Programming II
Fall 2019 - Programming Project 1
20 points

Out: September 9, 2019
Due: September 18, 2019 (Wednesday)

Reminder: Programming projects (unlike homework exercises) are to be your own work. See the syllabus for definitions of acceptable assistance from others.

SI session is on Mondays, 6-8pm, in KC-267. The instructor is Abdullah Aljandali (aa472).

Problem Statement
Boggle® is a game from the Hasbro Division of Parker Brothers. The game consists of a square grid filled with random letters. The goal of the game is to find as many words in the grid as possible. Here is an example grid:



The rules for forming words in Boggle® are as follows:

1. A word is formed by connecting a letter with any of its 8 neighbors. I.e., up, down, left, right, or diagonally.
2. An individual letter cannot be used more than once in a word.

For the example grid above, the word "settle" can be formed as shown above, but the word "fleet" cannot be formed. For this assignment, you will be given a 5x5 grid of lowercase letters and a list of words. Your task is to write a program to determine whether each word in the list can be formed in the grid using the Boggle® rules. The program file name must be `boggle.cpp`.

Coding Notes
This assignment is being distributed by the GitKeeper submission system. You should have received an email with the URL to clone. A reminder of the commands needed to set up this assignment:

```
$ cd <path to your class directory>
$ git clone <URL given in the assignment email>
```

This will create an working directory called **`project1-boggle`** containing example input file **`example.dat`**. Change to this directory and create **`boggle.cpp`**.

```
$ cd project1-boggle
$ emacs boggle.cpp &
```

Be sure to commit your changes early and often. **The first commit for this assignment is expected to be by Wednesday, September 11.** A reminder of the commands needed to do this (run in the working directory):

```
$ git add .
$ git commit -m "<description of what was completed>"
```

Implementation Requirements

The program must meet the following implementation requirements:

1. The program must accept **two command-line arguments** that are the names of an input file containing a grid of letters and a list of words and an output file containing the results. Appropriate checking of the number of command-line arguments and file opens is required. The format of the input file will be a 5 lines of 5 characters each forming the letter grid, followed by a list of words each on a separate line. The provided example input file **`example.dat`** contains the grid shown above and a list of words:

   ```
   npdnn
   avads
   itteb
   tplfa
   auemd
   pit
   pitted
   beads
   dates
   belt
   melted
   fame
   settle
   fleet
   fell
   belated
   added
   belted
   ```

2. The program should print out (just) the letter grid to the screen. I.e., if a border is used, the border should not be printed out. No other output should be printed to the screen. Any debugging code should be commented out before final submission.

3. The output file must have a line for each word consisting of the word, followed by a space, followed by **`found`** or **`not found`** (depending on whether the word can formed from the letters in the grid), followed by a newline. For the example above, the resulting output file would contain:

```
pit found
pitted found
beads found
dates found
belt found
melted found
fame found
settle found
fleet not found
fell not found
belated not found
added not found
belted not found
```

4. **Reading in the letter grid and printing out the letter grid must be done using functions.**

5. The problem of searching the grid for a word must be solved using a backtracking recursion technique similar to the maze program given in Lecture 8. Non-recursive solutions will be returned for resubmission and will incur a late penalty.

6. The grid must be implemented as a **two-dimensional array of structs** that contain a character (for the letter) and a boolean flag (to keep track of whether the letter has been used already). Unlike C, struct definitions introduce new type names and do not need the use of **`typedef`**. For example, the struct and array for this project can be defined using:

```
const int SIZE = 5;  // Size of the grid, 7 if using a border

struct Tile {
   char letter;
   bool visited;
};

// Declaration in main program
Tile grid[SIZE][SIZE];  // 2D grid of "letters"
```

   The identifier **`Tile`** becomes a type name that can be used to declare variables of the struct type as shown. Note that the type of the flag must be **`bool`**, not **`int`**, and the ending '**`;`**' is still needed at the end of the struct definition. Boolean variables must be assigned literal values **`true`** and **`false`** (i.e., do not use 1 and 0). Put this struct definition after any constant declarations and before the function prototypes.

   Note: the grid must be **declared inside the main program function and passed as a parameter** to the functions. Note: in C++, received-only multi-dimensional arrays should be passed const just like one-dimensional arrays. (This is not possible in C.)

7. A word in the word list must be stored in a C++ string, not a C string. The built-in C++ **`string`** type is defined in the **`<string>`** library header file. Single word strings are input and output like the primitive types using operators **`>>`** and **`<<`**, respectively, and the individual characters are accessed using **`[]`** like an array. In addition, C++ strings have a **`length`** operation that returns the number of characters in the string. E.g.,

```
#include <iostream>
#include <string>
int main () {
   using namespace std;
   string a_string;
   cout << "Enter a word: ";
   cin >> a_string;
   cout << a_string << " starts with character " << a_string[0]
        << ", and has " << a_string.length() << " characters" << endl;
   return 0;
}
```

Design Notes and Hints

- The main idea for the main program is to traverse the grid row by row, and for each letter, call the search function.
- The main idea for the search function is to keep track of which letter in the word you are looking for and compare it to the letter that is where you are in the grid. Think about how to represent these locations as parameters to the search function.
- There are several bases cases for stopping a search, one of which is when the word has been found, while the others are when you know that the word cannot be formed.
- Since a letter on an unfruitful path could be used later, it should be reset to unvisited when backtracking.
- Make sure you do not access outside the bounds of the actual letter grid. There are a number of different ways to do this.

REMINDER: Your project must compile to be graded. Projects that do not compile will receive a grade of 0. Submissions that do not substantially work will be returned for resubmission and assessed a late penalty.

Follow the program documentation guidelines in the *C++ Programming Style Guideline* handout. As stated in the syllabus, part of the grade on a programming project depends on how well you adhere to the guidelines. The grader will look at your code and grade it according to the guidelines.

How to submit

Please note that the automated submission system requires all files be named exactly as specified in an assignment (**`boggle.cpp`**, in this case), and also requires that the output of the program be exactly as expected including whitespace, capitalization, and spelling.

Assignments are submitted to GitKeeper by pushing committed changes.

```
$ git push
```

Only a clean repository (one where all changes have been committed) can be pushed. If a repository is not clean, git will say everything is up to date, and refuse to do the push. See handout *Using GitKeeper for Submitting Assignments* for additional information.