Fall 2019 CS 31

Programming Assignment 5
Flower Power

Time due: 9:00 PM Monday, November 18

You have been hired to write a program that plays the Flowers and Bees word guessing game. Here's how one round of the game works: The computer picks a mystery word of four to six letters and tells the player how many letters are in the word. The player tries to determine the mystery word by presenting the computer with a series of trial words. Each trial word is a four to six letter word. If the trial word is the mystery word, the player wins. Otherwise, the computer responds to the trial word with two integers: the number of flowers and the number of bees. Flowers and bees are pairings between a letter in the trial word and the same letter in the mystery word:

- A *flower* is a pairing of a letter in the trial word and the same letter in the mystery word in the same position. For example, if the mystery word is EGRET and the trial word is AGATE, there's one flower: The Gs in both words are in the same position, the second letter.
- A *bee* is a pairing between a letter in the trial word and the same letter in the mystery word, but *not* in the same position as in the trial word, provided that neither of the two letters are involved in a flower or another bee. For example, if the mystery and trial words are EGRET and AGATE, the Ts form a bee, since we can pair them up but they're not in the same position, since one is the fifth letter of EGRET and the other is the fourth letter of AGATE. The E in AGATE and, say, the first E in EGRET can be paired up to form another bee; the second E in EGRET would then not be part of a bee — we can't pair it up with the E in AGATE because that E is already paired up with the first E in EGRET. If instead we paired the E in AGATE with the second E in EGRET, then the first E in EGRET would have to remain unpaired.
- If a letter at a particular position in a word could be considered part of a flower or part of a bee, it must be treated as part of a flower. For example, if the mystery and trial words were EGRET and VIXEN, the E in VIXEN must be paired up with the second E in EGRET to form a flower; that takes priority over pairing it with the first E in EGRET to form a bee.

The player's score for each round is the number of trial words needed to determine the correct word (counting the trial word that matched the mystery word).

As an example, suppose the mystery word is EGRET. Here are some examples of the flower and bee counts for various trial words:

```
  EGRET        EGRET        EGRET        EGRET        EGRET
  GOOSE        RAGE         SIREN        EERIE        EERIE
  b   b        b bf           ff         fbf     or   f f b
0 f / 2 b    1 f / 2 b    2 f / 0 b    2 f / 1 b    2 f / 1 b

  EGRET        EGRET        EGRET        EGRET
  GREET        OKAPI        REGRET  or   REGRET
  bbbff                     bbb bb        bbbbb
2 f / 3 b    0 f / 0 b    0 f / 5 b    0 f / 5 b
```

Your program must ask the player how many rounds to play, and then play that many rounds of the game. After each round, the program must display some statistics about how well the player has played the rounds so far: the average score, the minimum score, and the maximum score.

Here is an example of how the program must interact with the player (player input is in **boldface**):

```
How many rounds do you want to play? 3
```

```
Round 1
The mystery word is 5 letters long.
Trial word: assert
Flowers: 1, Bees: 2
Trial word: xyzzy
I don't know that word.
Trial word: bred
Flowers: 0, Bees: 2
Trial word: mucus
Flowers: 0, Bees: 0
Trial word: never
Flowers: 2, Bees: 2
Trial word: enter
Flowers: 1, Bees: 2
Trial word: river
Flowers: 3, Bees: 0
Trial word: raven
You got it in 7 tries.
Average: 7.00, minimum: 7, maximum: 7

Round 2
The mystery word is 5 letters long.
Trial word: eerie
Flowers: 2, Bees: 1
Trial word: rage
Flowers: 1, Bees: 2
Trial word: greet
Flowers: 2, Bees: 3
Trial word: egret
You got it in 4 tries.
Average: 5.50, minimum: 4, maximum: 7

Round 3
The mystery word is 4 letters long.
Trial word: monkey
Flowers: 0, Bees: 0
Trial word: puma
Flowers: 0, Bees: 0
Trial word: Hello
Your trial word must be a word of 4 to 6 lower case letters.
Trial word: what?
Your trial word must be a word of 4 to 6 lower case letters.
Trial word: wrap-up
Your trial word must be a word of 4 to 6 lower case letters.
Trial word: stop it
Your trial word must be a word of 4 to 6 lower case letters.
Trial word: sigh
You got it in 3 tries.
Average: 4.67, minimum: 3, maximum: 7
```

Notice that unknown words and trial strings that don't consist of exactly 4 to 6 lower case letters don't count toward the number of tries for a round.

You can assume the player will always enter an integer for the number of rounds (since you haven't learned a clean way to check that yet). If the number of rounds entered is not positive, write the message

```
The number of rounds must be positive.
```

(not preceded by an empty line) to `cout` and terminate the program immediately.

The program will be divided across three files: `flowers.cpp`, which you will write; `utilities.h`, which we have written and which you must not change; and `utilities.cpp`, which we have written and you must not change.

You will turn in only `flowers.cpp`; when we test your program, our test framework will supply `utilities.h` and our own special testing version of `utilities.cpp`.

In order for us to thoroughly test your program, it must have at least the following components:

- In `flowers.cpp`, a main routine that declares an array of C strings. This array exists to hold the list of words from which the mystery word will be selected. The response to a trial word will be the number of flowers and bees only if the trial word is in this array. (From the example transcript above, we deduce that "xyzzy" is not in the array.) The declared number of C strings in the array must be at least 9000. (You can declare it to be larger if you like, and you don't have to use every element.)

  Each element of the array must be capable of holding a C string of length up to 6 letters (thus 7 characters counting the zero byte). So a declaration such as `char wordList[9000][7];` is fine, although something like `char wordList[MAXWORDS][MAXWORDLEN+1];`, with the constants suitably defined, would be stylistically better.

  Along with the array, your main routine must declare an int that will contain the actual number of words in the array (i.e., elements 0 through one less than that number are the elements that contain the C strings of interest). The number may well be smaller than the declared size of the array, because for test purposes you may not want to fill the entire array.

  Before prompting the player for the number of rounds to play, your main routine must call `getWords` (see below) to fill the array. The only valid words in the game will be those that `getWords` puts into this array.

  If the player's score for a round is not 1, the message written to cout reporting the score must be

  ```
  You got it in n tries.
  ```

  where *n* is the score. If the score is 1, the message must be

  ```
  You got it in 1 try.
  ```

- In `utilities.cpp`, a function named `getWords` with the following prototype:

  ```
  int getWords(char words[][7], int maxWords, const char wordfilename[]);
  ```

  (Instead of `7`, our `utilities.cpp` actually says `MAXWORDLEN+1`, where `MAXWORDLEN` is declared to be the constant 6 in `utilities.h`.) This function puts words into the `words` array and returns the number of words put into the array. The array must be able to hold at least `maxWords` words. The file named by the third argument is the plain text file that contains the words, one per line, that will be put into the array.

  You *must* call `getWords` exactly once, before you start playing any of the rounds of the game. If your main routine declares `wordList` to be an array of 10000 C strings and `nWords` to be an int, you'll probably invoke this function like this:

  ```
  const char WORDFILENAME[] = "the path for the word file";
  ...
  int nWords = getWords(wordList, 10000, WORDFILENAME);
  ```

  You may use this 7265-word file if you want a challenging game. Here's how you'd specify the path to the word file for various systems:

  - Windows: Provide a path for the filename, but use / in the string instead of the \ that Windows paths use, e.g. `"Z:CS31/P5/mywordfile.txt"`.
  - Mac: It's probably easiest to use the complete pathname to the words file, e.g. `"/Users/yourUsername/words.txt"` or `"/Users/yourUsername/CS31/P5/words.txt"`.

  - Linux: If you put the words.txt file in the same directory as your .cpp file, you can use `"words.txt"` as the file name string.

We have given you an implementation of `getWords`. (Don't worry if you don't understand every part of the implementation.) It fills the array with the four-to-six-letter words found in the file named as its third argument. To do simple testing, you can *temporarily* change the implementation of `getWords` to something like this that ignores the file name and hard codes a small number of words to be put in the array:

```
int getWords(char words[][7], int maxWords, const char wordfilename[])
{
    if (maxWords < 2)
        return 0;
    strcpy(words[0], "eagle");
    strcpy(words[1], "goose");
    return 2;
}
```

Whatever implementation of `getWords` you use, each C string that it puts into the array must consist of four to six lower case letters; the C strings must not contain any characters that aren't lower case letters. If you have made a temporary change to `getWords` for test purposes, be sure to restore `utilities.cpp` back to its original state and verify that your program still runs correctly.

The `getWords` function must return an int no greater than `maxWords`. If it returns a value less than 1, your main routine must write

```
No words were loaded, so I can't play the game.
```

to `cout` and terminate the program immediately, without asking the player for the number of rounds to play, etc.

When we test your program, we will replace `utilities.cpp` (and thus any changed implementation of `getWords` you might have made) with our own special testing implementation that will ignore the third argument and fill the array with the test words we want to use.

If `getWords` returns a value in the range from 1 to `maxWords` inclusive, your program must write no output to`cout` other than what is required by this spec. If you want to print things out for debugging purposes, write to `cerr` instead of `cout`. When we test your program, we will cause everything written to `cerr` to be discarded instead — we will never see that output, so you may leave those debugging output statements in your program if you wish.

- In `flowers.cpp`, a function named `playOneRound` with the following prototype:

```
int playOneRound(const char words[][7], int nWords, int wordnum);
```

(Again, instead of `7`, you can use something like `MAXWORDLEN+1`.) Using `words[wordnum]` as the mystery word, this function plays one round of the game, reading from cin and writing to cout. It returns the score for that round. In the transcript above, for round 1, for example, this function is responsible for this much of the round 1 output, no more, no less:

```
Trial word: assert
Flowers: 1, Bees: 2
Trial word: xyzzy
I don't know that word.
Trial word: bred
Flowers: 0, Bees: 2
Trial word: mucus
Flowers: 0, Bees: 0
Trial word: never
Flowers: 2, Bees: 2
Trial word: enter
```

```
        Flowers: 1, Bees: 2
        Trial word: river
        Flowers: 3, Bees: 0
        Trial word: raven
```

Your program must call this function to play each round of the game. Notice that this function does *not*select a random number and does *not* tell the user the length of the mystery word; the *caller* of this function does, and passes the random number as the third argument. Notice also that this function does *not* write the message about the player successfully determining the mystery word. **If you do not observe these requirements, your program will fail most test cases.**

The parameter `nWords` represents the number of words in the array; if it is not positive, or if `wordnum` is less than 0 or greater than or equal to `nWords`, then `playOneRound` must return −1 without writing anything to `cout`.

If for a trial word the player enters a string that does not contain four to six lower case letters or contains any character that is not a lower case letter, the response written to `cout` must be

```
        Your trial word must be a word of 4 to 6 lower case letters.
```

If the player enters a string consisting of exactly four to six lower case letters, but that string is not one of the words in the array of valid words, then the response written to `cout` must be

```
        I don't know that word.
```

To make things interesting, your program must pick mystery words at random using the function `randInt`, contained in `utilities.cpp`:

```
        int randInt(int min, int max);
```

Every call to `randInt` returns a random integer between min and max, inclusive. If you use it to generate a random position in an array of `n` interesting items, you should invoke it as `randInt(0, n-1)`, not `randInt(0, n)`, since the latter might return `n`, which is not a valid position in an `n`-element array.

Your program must *not* use any `std::string` objects (C++ strings); you must use C strings.

You may assume (i.e., we promise when testing your program) that any line entered in response to the trial word prompt will contain fewer than 100 characters (not counting the newline at the end).

Your program must **not** use any global variables whose values may change during execution. Global *constants* are all right; our `utilities.h` declares `const int MINWORDLEN = 4;` globally, for example, and it's fine for your `flowers.cpp` to declare your own additional global constants. The reason for this restriction is that part of our testing will involve replacing your `playOneRound` function with ours to test some aspects of your `main` function, or replacing your `main` with ours to test aspects of your `playOneRound`. For this reason, you must not use any non-const global variables to communicate between these functions, because our versions won't know about them; all communication between these functions must be through the parameters (for `main` to tell `playOneRound` the words, number of words, and mystery word number for a round), and the return value (for `playOneRound` to tell `main` the score for that round). Global *constants* are OK because no function can change their value in order to use them to pass information to another function.

Microsoft made a controversial decision to issue by default an error or warning when using certain functions from the standard C and C++ libraries (e.g., `strcpy`). These warnings call that function unsafe and recommend using a different function in its place; that function, though, is not a Standard C++ function, so will cause a compilation failure when you try to build your program under clang++ or g++. Therefore, for this class, we do not want get that error or warning from Visual C++; to eliminate them, put the following line in your program *before* any of your `#includes`:

```
#define _CRT_SECURE_NO_DEPRECATE
```

It is OK and harmless to leave that line in when you build your program using clang++ or g++.

Visual C++ gives a harmless warning if you declare a large array (like the list of words) in a function; the warning starts out `Function uses '`*`NNNNN`*`' bytes of stack: exceeds /analyze:stacksize '16384'`. To get rid of it, put the following line in your program above the `#define _CRT_SECURE_NO_DEPRECATE`:

```
#pragma warning(disable:6262)
```

It is OK and harmless to leave that line in when you build your program using clang++ or g++, even if you get a warning about the pragma being ignored.

What you will turn in for this assignment is a zip file containing these two files and nothing more:

1. A text file named **flowers.cpp** that contains `main`, `playOneRound`, and other functions you choose to write that they might call. (You must *not* put implementations of `getWords` or `randInt` in `flowers.cpp`; they belong in `utilities.cpp`.) Your source code should have helpful comments that tell the purpose of the major program segments and explain any tricky code.

2. A file named **report.docx** or **report.doc** (in Microsoft Word format), or **report.txt** (an ordinary text file) that contains:
   a. A brief description of notable obstacles you overcame.
   b. A description of the design of your program. You should use pseudocode in this description where it clarifies the presentation. Document the design of your main routine, `playOneRound`, and any other functions you write. Do not document the `getWords` or `randInt` functions.

   Your report does not need to describe the data you might use to test this program.

By November 17, there will be links on the class webpage that will enable you to turn in your zip file electronically. Turn in the file by the due time above.

Additional requirements

A few days before the project is due, we will provide a test procedure that you can follow to check whether you are making a foolish, yet easy to fix mistake that could cause you to fail a lot of our test cases. These mistakes include formatting or spelling differences from what this spec shows or your having `playOneRound` do or write something that your main routine must instead, or vice versa.

Standard C++ requires that the number of elements in an array you declare to be known at compile time. Since the g31 command on cs31.seas.ucla.edu enforces that requirement, and your program must run under that compiler, you must meet that requirement. Thus, you must not do something like this:

```
void somefunction(char someword[])
{
    char a[strlen(someword)+1]; // Error! strlen(someword) not known at compile time
```

Getting started

You've already learned from Project 3 how to set up a program with multiple source files. This project is also having you do someting we haven't done before: run a program that reads from a file. Before you delve into the details of writing the code to play the game, you would be wise to ensure that you can do this new things correctly. We will first have you set up a file with a couple of words in it that your test program will read. Then you will set up a project and run the program to read the file.

First, place this two-word file at a location of your choosing. On a Windows or Mac system, make sure you know the complete path name to the file (e.g., `Z:CS31\P5\mywordfile.txt` or `C:\Temp\smallwords.txt` on a Windows system, or `/Users/yourUsername/CS31/P5/smallwords.txt` on a Mac). Then, set up a project consisting of our`utilities.cpp`, our `utilities.h`, and `flowers.cpp`. For the text of the `flowers.cpp` file, use this:

```
#include "utilities.h"
#include <iostream>
using namespace std;

const char WORDFILENAME[] = "the path to your file of words";

int main()
{
    char w[9000][7];
    int n = getWords(w, 9000, WORDFILENAME);
    if (n == 2)
    {
        cout << "getWords successfully found the file and read its two words." << endl;
        cout << "You're ready to start working on Project 5." << endl;
    }
    else if (n == -1)
        cout << "The path to your file of words is probably incorrect" << endl;
    else
        cout << "getWords found the file, but loaded " << n
             << " words instead of 2" << endl;
}
```

Running the project using g31 with Linux

To take a three-file project you've developed with Visual C++ or Xcode and run it with g31 under Linux, follow these steps.

1. Follow steps 1 through 3 of the g++ with Linux writeup to transfer the three files (`utilities.h`,`utilities.cpp`, and `flowers.cpp`) to the Windows desktop on a SEASnet machine and to log in to `cs31.seas.ucla.edu`.

2.

3. Create a new directory for this project; let's call it `proj5`:

   ```
   mkdir proj5
   ```

4. Copy the files from the Desktop to this directory:

   ```
   cp Desktop/utilities.* Desktop/flowers.cpp proj5
   ```

5. Make `proj5` the *current directory* (i.e., the default directory for now in which files will be found or created):

   ```
   cd proj5
   ```

6. Verify that the expected three files are present by listing the contents of the current directory:

   ```
   ls
   ```

7. Copy over the two-word word file into the current directory:

   ```
   curl -s -L http://cs.ucla.edu/classes/fall19/cs31/Projects/5/smallwords.txt > twowords.txt
   ```

8. Use the Nano editor to change the initializer for `WORDFILENAME` to `"twowords.txt"`.

   ```
   nano flowers.cpp
   ```

You can navigate with the arrow keys. The bottom two lines of the display show you some commands you can type. For example, control-O (indicated in the bottom display as `^O`) saves any changes you make to the file, and control-X exits the editor.

9. Build an executable file from the source files. If we would like the executable file to be named `testflowers`, we'd say

   ```
   g31 -o testflowers *.cpp
   ```

   The `*.cpp` saves us typing individual file names by matching all the files whose names end in `.cpp`. (Notice that we do not list the .h file.).

10. To execute the program `testflowers` that you built, you'd just say

    ```
    ./testflowers
    ```