Programming Assignment 5
Star Search

Time due: 9:00 PM Monday, November 20

You have been contracted by the producers of the highly-rated Star Search game show to write a program that lets fans play a home version of the game. Here's how one round of the game works: The computer picks a secret word of four to six letters and tells the player how many letters are in the word. The player tries to determine the secret word by presenting the computer with a series of probe words. Each probe word is a four to six letter word. If the probe word is the secret word, the player wins. Otherwise, the computer responds to the probe word with two integers: the number of stars and the number of planets. Stars and planets are pairings between a letter in the probe word and the same letter in the secret word. A *star* is a pairing of a letter in the probe word and the same letter in the secret word in the same position. A *planet* is a pairing between a letter in the probe word and the same letter in the secret word, but *not* in the same position as in the probe word, provided that neither of the two letters are involved in a star or another planet. The player's score for each round is the number of probe words needed to get the correct word (counting the probe word that matched the secret word).

(The terms *star* and *planet* come from the astronomical objects; stars have a fixed position in the sky relative to other stars, while planets appear to wander relative to the stars.)

As an example, suppose the secret word is EGRET. Here are some examples of the star and planet counts for various probe words:

```
   EGRET        EGRET        EGRET        EGRET
   GOOSE        RAGE         EERIE        EERIE
   p   p        p ps         sps     or   s s p
 0 s / 2 p    1 s / 2 p    2 s / 1 p    2 s / 1 p

   EGRET        EGRET        EGRET        EGRET
   GREET        OKAPI        REGRET  or   REGRET
   pppss                     ppp pp        ppppp
 2 s / 3 p    0 s / 0 p    0 s / 5 p    0 s / 5 p
```

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 secret word is 5 letters long.
Probe word: assert
Stars: 1, Planets: 2
Probe word: xyzzy
I don't know that word.
Probe word: bred
Stars: 0, Planets: 2
Probe word: mucus
Stars: 0, Planets: 0
Probe word: never
Stars: 2, Planets: 2
Probe word: enter
Stars: 1, Planets: 2
Probe word: river
Stars: 3, Planets: 0
Probe word: raven
You got it in 7 tries.
Average: 7.00, minimum: 7, maximum: 7

Round 2
The secret word is 5 letters long.
Probe word: eerie
Stars: 2, Planets: 1
Probe word: rage
Stars: 1, Planets: 2
Probe word: greet
Stars: 2, Planets: 3
Probe word: egret
You got it in 4 tries.
Average: 5.50, minimum: 4, maximum: 7

Round 3
The secret word is 4 letters long.
Probe word: monkey
Stars: 0, Planets: 0
Probe word: puma
Stars: 0, Planets: 0
Probe word: Hello
Your probe word must be a word of 4 to 6 lower case letters.
Probe word: what?
Your probe word must be a word of 4 to 6 lower case letters.
Probe word: wrap-up
Your probe word must be a word of 4 to 6 lower case letters.
Probe word: stop it
Your probe word must be a word of 4 to 6 lower case letters.
Probe word: sigh
```

```
You got it in 3 tries.
Average: 4.67, minimum: 3, maximum: 7
```

Notice that unknown words and probe 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) and terminate the program immediately.

The program will be divided across three files: `stars.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 `stars.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 `stars.cpp`, a main routine that declares an array of C strings. This array exists to hold the list of words from which the secret word will be selected. The response to a probe word will be the number of stars and planets only if the probe 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 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`, you can use something like `MAXWORDLEN+1`, where `MAXWORDLEN` is declared to be the constant 6, as 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 `stars.cpp`, a function named `runOneRound` with the following prototype:

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

  (Again, instead of `7`, you can use something like `MAXWORDLEN+1`.)
  Using `words[wordnum]` as the secret word, this function plays one round of the game. 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:

  ```
  Probe word: assert
  Stars: 1, Planets: 2
  Probe word: xyzzy
  I don't know that word.
  Probe word: bred
  Stars: 0, Planets: 2
  Probe word: mucus
  Stars: 0, Planets: 0
  Probe word: never
  Stars: 2, Planets: 2
  Probe word: enter
  Stars: 1, Planets: 2
  Probe word: river
  Stars: 3, Planets: 0
  Probe 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 secret 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 secret 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 `runOneRound` must return −1 without writing anything to `cout`.

  If for a probe 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 must be

  ```
  Your probe 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 must be

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

To make things interesting, your program must pick secret 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 probe 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; it's perfectly fine to declare `const int MINWORDLEN = 4;` globally, for example. The reason for this restriction is that part of our testing will involve replacing your `runOneRound` function with ours to test some aspects of your `main`function, or replacing your `main` with ours to test aspects of your `runOneRound`. 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 `runOneRound` the words, number of words, and secret word number for a round), and the return value (for `runOneRound` 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 `#include`s:

```
        #define _CRT_SECURE_NO_DEPRECATE
```

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

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

1. A text file named **stars.cpp** that contains `main`, `runOneRound`, and other functions you choose to write that they might call. (You must *not* put implementations of `getWords` or `randInt` in `stars.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 and any 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 19, 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.

Getting started

This project is having you do two things we haven't done before: set up a project with multiple source files and 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 these two 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, follow the instructions below for the compiler you'll be using. For the text of the `stars.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;
        }
```

Setting up a multi-file Visual C++ project

Follow these steps to set up a Visual C++ project to compile and run a three-file program. A number of these steps may have alternate ways to achieve the same end.

1. Follow steps 1 through 4 in the Visual C++ 2017 writeup to create an empty Visual C++ project.
2. From the Project menu, select Add New Item. In the Add New Item dialog, select C++ file (.cpp) in the middle panel. Enter the source file name `stars`, in the Name text box below. Click Add.
3. Edit the `stars.cpp` file in the window that appears. Insert the code in the Getting started section. For the initializer for `WORDFILENAME`, replace any \ in the pathname to your words file with / (e.g., use `"Z:CS31/P5/mywordfile.txt"` or `/Temp/smallwords.txt"`).
4. From the Project menu, select Add New Item. This time, in the Add New Item dialog, select Header file (.h) in the middle panel and enter the header file name `utilities` in the Name text box below. Copy and paste our utilities.h into the `utilities.h` file that appears.
5. From the Project menu, select Add New Item. This last time, select C++ file (.cpp) in the middle panel, and enter the source file name `utilities` in the Name text box below. Copy and paste our utilities.cpp into the `utilities.cpp` file that appears.
6. From the Debug menu, select Start Without Debugging. This will save any changed files, compile the source files, and run the resulting executable if there were no build errors. (If you select Start Debugging, your console window screen will disappear as soon as your program finishes executing, which you don't want.)

Setting up a multi-file Xcode project

Follow these steps to set up an Xcode project to compile and run a three-file program. A number of these steps may have alternate ways to achieve the same end.

1. Follow steps 1 through 4 in the Xcode on a Mac writeup to create a project. Use `stars` as the project name instead of `hello`.
2. In the left frame, click once on the name `main.cpp` to open the file. Click once again on the name to rename the file to `stars.cpp`. Once you've clicked outside the text box to effect the name change, replace the code that Xcode supplied with the code in the Getting started section. For the initializer for `WORDFILENAME`, use the pathname to your words file (e.g. `/Users/`*`yourUsername`*`/CS31/P5/smallwords.txt`).
3. In the left frame, right click on the yellow folder icon labelled "stars". In the popup menu, select New File. Select the macOS tab, and in the Source group, select C++ File. Click Next. Use `utilities` for the name and leave the box checked for also creating a header file. Click Next. A save file dialog appears; click Create. Rename the file that was created named `utilities.hpp` to be named `utilities.h`. Replace the code in `utilities.h` with our utilities.h. Replace the code in `utilities.cpp` with our utilities.cpp.
4. Follow steps 6 through 9 in the Xcode on a Mac writeup to build and run your program.

Running the project using g++ with Linux

To take a three-file project you've developed with Visual C++ or Xcode and run it with g++ 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 `stars.cpp`) to the Windows desktop on a SEASnet machine and to log in to `cs31.seas.ucla.edu`.
2. Create a new directory for this project; let's call it `proj5`:

   ```
   mkdir proj5
   ```

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

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

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

   ```
   cd proj5
   ```

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

```
  ls
```

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

```
  curl -s -L
http://cs.ucla.edu/classes/fall17/cs31/Projects/5/smallwords.txt >
twowords.txt
```

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

```
  nano stars.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.

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

```
  g31 -o teststars *.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.).

9. To execute the program `teststars` that you built, you'd just say

```
  teststars
```

   (If that doesn't work, try `./teststars`)