The Password Game (Using C++)

First, the user shall launch the entire program with a single command. At launch time, the first command line argument (after the program name) will indicate the name of an ASCII coded text file to load in the current directory. That is, we assume that the file name specified at launch time will exist in the same directory as the program. For example, the user may call your program like so:

```
passwordgame JaneAustin.txt
```

The program will verify the file exists and that it can open it. If not, it shall warn the user of the missing file and terminate the program. If the user fails to supply enough arguments at run-time, the program shall display a helpful use prompt and exit. Under no circumstances shall the program use a default, local file hard-coded in the software. Students frequently do this just to "get things running," and they then submit the code with the local file path. Our machines do not have access to c:\users\spongebob\my documents\programs\cs320\crabby.txt, so the program fails at runtime. Avoid this problem by **never** hard-coding a magic file name. Use the command line arguments from the get go, and you will not have problems.

After loading the file provided at program launch, the program shall then prompt the user in real-time for the number of tokens to use when generating passwords. The user may enter anything from 1-5. This number represents the number of words it will combine to build a password.

Using the file designated at program launch, the program shall identify all the unique words it holds, and it will then randomly combine them to build a password of the specified length. This is the phrase that the computer will then attempt to brute-force solve with other functions. That is, it will pretend it doesn't already know the answer, and then start guessing passwords until it successfully matches the one it generated already.

The program will attempt to guess the password using two approaches in multiple threads. Both processes know how many words the password uses, so it doesn't need to build up guesses, and it can start generating passwords containing the correct number of combined words. The first approach shall simply combine words together randomly until it detects a match. The second approach will begin sequentially walking through the words changing only one at a time until it exhausts every combination.

Because the password generator and password guessers use the same dictionary, the program will definitely guess the password at some point.

Which of these approaches do you think will perform better?

Additional details for each component follows.

Unique Token Detector

Students must create a class/struct (or function) that parses the input file and stores every unique token it finds. A token is any white-space delineated word. The software must strip any trailing punctuation and discard capitalization. For example, given the following quote from RADM Grace Hopper:

"Leadership is a two-way street, loyalty up and loyalty down."

The software will detect the following unique tokens:

```
["leadership", "is", "a", "two-way", "street", "loyalty", "up", "and", "down"]
```

Having detected these tokens, the function or class shall return them as a std::list of strings. Programs need not sort this list, but it may prove helpful later on. Students may use std::string for this task. If students implement this portion as a class, they should specify a public method named getUnique that returns these items.

Password Generator

Teams shall construct a class or struct (**not** a function) that accepts a std::list of std::string objects and uses them to generate passwords when requested. This class shall include the following public methods:

PasswordGenerator(std::list<std::string>); // A public constructor with the unique tokens to use

std::string getRandomPassword(int numWords); // returns a randomly generated password from the stored list using numWords from the list. The function shall include a space between words. No two words may be the same (no duplicates!)

void setIterationLength(int numWords); // when sequentially generating words, this establishes how many to return when the next() method is called. This resets the class' iteration to the beginning.

std::string next(); // using the length of iteration established by the setIterationLength, this function shall sequentially step through every single combination of that length (in order). If using a word length of one, the method shall simply step through the list of unique tokens it received. For a numWords of 2, the method shall fix the first word at the

starting word, and then supply every other possible second word from the list (in order). When that completes, it shall increment the first word to the next word and do it all again. This is a SLOW, brute force approach. For passwords with numWords 4, this will take a while.

bool hasNext(); // returns true if the iteration has not reached its end. (there are more combinations to try!)

Students may include any reasonable number of additional private methods to simplify their code and make it more readable.

Password Guesser

This class shall use a PasswordGenerator to establish a random password with the number of tokens equal to what the user specified during program launch. It shall then create two threads which will each begin guessing passwords (they must obviously know the password they're trying to break, so it should arrive as a parameter). The first thread shall use the PasswordGenerator's iterator to step through each combination while the second thread simply keeps requesting new random passwords from the generator until one matches. I recommend using the observer design pattern here, for the moment one of them finds the password, the other no longer needs to search. Have the Password Guesser loop or sleep until one of them call's its notify method

Extra Credit

Teams may earn additional credit through excellent performance. Displaying information in clean and formatted output about the input file (e.g., number of unique tokens) and the password combinations (e.g., given the input file tokens and the specified length, the program can build N different password combinations) is one way.

Timing the two tasks and displaying (not spamming) updates as the two guessing routines runs is another way to earn the extra credit points. Think about using the observer design pattern and including the timing information as one of the parameters in the subscriber's update method.

In summary, a clean, professional solution (at the grader's discretion) earns extra credit.