Summer 2015 CS 31

Programming Assignment 3
Spamalot

Time due: 9:00 PM Tuesday, July 14

Introduction

Jee-mail, a small email service provider, has been receiving complaints from customers claiming that their inboxes are clogged with the dozens of spam email messages they receive each day. Consequently, Jee-mail has contracted with the SmallSoft Corporation to design and build a spam detection system. Of course, since SmallSoft gets all of its labor by exploiting UCLA undergraduate students, you'll be doing all of the programming.

Given one or more email messages, your program will have to classify each one as either spam or legitimate. Jee-mail wants you to build a point-based anti-spam system. Your program is to look for certain suspicious features in the email, such as frequent use of exclamation points (!), use of words like*FREE*, or excessive use of uppercase letters. Each time such a suspicious feature is found, you increase a score that indicates how suspicious an email is. If an email's score exceeds 100 points, then it is classified as spam; otherwise, it is deemed legitimate. The detailed rules for scoring a message are in the Spam Rules section below.

Sample Transcript

Here is a sample transcript of the program, with **boldface** text representing the user's input.

```
Enter the subject line of the email: MAKE MONEY FAST XTFWAQQQ
Enter the body of the email.  Press Enter on an empty line to finish.
THIS IS YOUR CHANCE TO MAKE lots of moolah!
CLICK ON www.lotsofmoolah.com FOR MORE INFORMATION!
        ← the user just hit the enter key on this line
This email is classified as spam, because its spam score is 115.
Would you like to classify another email (y or n)? Quit
Please enter y or n.
Would you like to classify another email (y or n)? Y
Please enter y or n.
```

```
Would you like to classify another email (y or n)? y
Enter the subject line of the email: hi mom
Enter the body of the email.  Press Enter on an empty line to finish.
Hi mom,
I hope you're doing OK.  My CS class is great, but the instructor is WEIRD!
I love you!
        ← the user just hit the enter key on this line
This email is classified as legitimate, because its spam score is 0.
Would you like to classify another email (y or n)? n

Number of spam messages: 1
Number of legitimate messages: 1
```

Input and Output Specification

The first line that your program must print out is exactly this (with one space after the colon):

```
Enter the subject line of the email:
```

Your program must then read the subject line the user types in. (It is not an error if the subject line is the empty string.) After that, your program must print out exactly this line:

```
Enter the body of the email.  Press Enter on an empty line to finish.
```

Your program must then read the message body lines the user types in. The message body consists of zero or more lines. After your program reads an empty line, which indicates the end of the message, it must print out exactly one of these two messages, depending on whether applying the spam rulesclassified the message as spam or not:

```
This email is classified as spam, because its spam score is number.
```

or

```
This email is classified as legitimate, because its spam score is number.
```

In these messages, *number* is to be replaced by the appropriate spam score, such as `115`, or `0`, or `5`.

After printing the classification message, your program must print exactly this (with one space after the question mark):

```
Would you like to classify another email (y or n)?
```

Your program must then read the line the user types in. If the user types only a lowercase `y`, the program must process another email, starting with the step where it prints the prompt for the subject line of the email.

If instead, the user types only a lowercase `n`, the program must print an empty line, then print the following two lines, then terminate:

```
Number of spam messages: number
Number of legitimate messages: number
```

In these messages, the first *number* is the number of messages that were classified as spam, and the second is the number that were classified as legitimate.

If the user types anything other than a single lowercase `y` or `n` in response to the question, the program must print out exactly the line:

```
Please enter y or n.
```

and repeat the prompt about classifying another email. It must repeatedly prompt the user until `y` or `n` is selected.

Spam Rules

The Jee-mail spam experts have come up with five rules, each of which can contribute some points to the spam score of a message. If the total spam score for the message is more than 100, the message is classified as spam; otherwise, it is considered a legitimate message. You must use only these five rules; do not improvise and make up your own. (Save that for when you start your own competing spam filter company.)

First, some definitions: a *word* in a string is a contiguous sequence of letters delimited either by non-letter characters or by the start or the end of the string. A letter may be upper- or lowercase. As an example, the string `It's like the old-fashioned 1980s!` has these seven words: `It s like the old fashioned s`. A *consonant* is a letter other than one of these ten:`aeiouAEIOU`.

Here are the five rules. The first three relate to the subject line of the email, while the last two are concerned with the body of the message.

1. If the subject line has at least one word, and *more than* 90% of the words in the subject line consist of only uppercase letters, then add 30 points to the spam score. For example, a subject line of

```
GET A DIPLOMA FREE!!
```

meets this criterion.

2. If the *last word* of the subject line contains *more than* 3 consecutive consonants, then add 40 points to the spam score. Spammers often place random words at the end of a subject line to confuse some anti-spam software; this rules detects many of those random words. For example, this subject line earns the 40 point addition to the score:

   ```
   This stock is going up, up, up!  mqafrpxo!
   ```

3. If the subject line contains 3 or more contiguous exclamation points, then add 20 points to the spam score. This subject line qualifies for the 20 points, for example:

   ```
   Make money fast!!!  No experience necessary!!!!!
   ```

4. If the *body* of the message has at least one word, and *more than* 50% of the words making up the body of the email consist entirely of uppercase letters, then add 40 points to the score. For example, this email:

   ```
   Subject:  Greetings from your good friend
   Body:
           THIS IS YOUR CHANCE TO MAKE lots of moolah!
           GO TO www.lotsofmoolah.com FOR MORE INFORMATION!
   ```

   earns the 40 points because there are 17 words in the body of the email, 11 of which (64.7%) are entirely uppercase.

5. *Each time* one of the following special words is found in the *body* of the email, add 5 points to the spam score for that message. The letters in these words may be in either case (e.g., `Sex` is the same as `sex`):

   ```
   buy
   cheap
   click
   diploma
   enlarge
   free
   lonely
   money
   now
   offer
   only
   pills
   ```

```
sex
```

For example, this message would score 35 points, 5 for each of the words indicated in boldface:

```
Subject:  Looking for cheap tickets?
Body:
        You can find cheap! tickets ONLY on our website.  Save money with this
        great OfFeR!!  Don't be a cheapskate.  Click now for this offer!
```

Notice that the "cheap" in the subject line is ignored by this rule, as is the word "cheapskate" in the second line of the body.

Functions You Must Write

We know that this will be the most complex program that many of you will have written in your programming careers to date, so we will make it easier for you by decomposing the problem for you.

To get full credit for this project, you **must** implement all of the following functions, using the exact function names, parameters types, and return types shown in this specification. The functions must not use any global variables whose values may change. (Global *constants* are all right.) In addition to testing your program as a whole, we will also test each of the following functions separately. That way, if your program doesn't work overall but you correctly implemented some of the functions, you'll still get some credit.

The functions you must implement are

```
getFirstWord
getLastWord
extractWord
isUppercase
makeUppercase
hasMultipleExclamations
isGibberishWord
```

Your solution should use these functions where appropriate (although you might not use every one of them). One reason is to help you get partial credit in the case where most of your program is correct, but a few of the function implementations are incorrect.

To help you implement these functions, you might choose to write additional functions as well. While we won't test those

additional functions separately, using them may help you structure your program more readably. If you find it helpful, you might have some of these seven required functions call other functions in that list.

When you turn in your solution, none of these seven required functions, nor any functions that they call, may read any input from `cin` or write any output to `cout`. (Of course, during development, you may have them write whatever you like to help you debug.) If you want to print things out for debugging purposes, write to `cerr` instead of `cout`. `cerr` is the standard error destination; items written to it by default go to the screen. 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.

string getFirstWord(string text)

This function returns the first word in the string variable `text`. If `text` has no words, this function returns the empty string (i.e. ""). Here is an incomplete test driver for this function:

```
void test()
{
      // This writes "Hello"
    cerr << getFirstWord("!!Hello, Fred") << endl;

      // This writes "greetings 9"
    string msg = "greetings, mom, how are you?";
    string result = getFirstWord(msg);
    cerr << result << " " << result.size() << endl;

      // This writes "0"
    string s = getFirstWord(" $@#%!!");
    cerr << s.size() << endl;
}
```

Hint: The string `substr` function may be of use in the implementation of this function.

string getLastWord(string text)

This function returns the last word in the string variable `text`. If `text` has no words, this function returns the empty string (i.e. ""). As an example,`getLastWord("MAKE MONEY FAST!!")` returns `"FAST"`.

Hint: The string `substr` function may be of use in the implementation of this function.

string extractWord(string& text)

This function returns the first word in the string that `text` is a reference to. If that string has no words, this function returns the empty string.

This function also modifies the string that `text` refers to. If that string has no words, then this function sets `text` to the empty string. Otherwise, it removes from `text` the first word of `text` and all non-letters preceding that first word. Here is an incomplete test driver:

```
void test()
{
    string s = "***AMAZING!*** Do it, now!!";
    string w = extractWord(s);
      // This writes "AMAZING" and "!*** Do it, now!!"
    cerr << w << endl << s << endl;

    w = extractWord(s);
      // This writes "Do" and " it, now!!" (space before "it")
    cerr << w << endl << s << endl;

    w = extractWord(s);
      // This writes "it" and ", now!!"
    cerr << w << endl << s << endl;

    w = extractWord(s);
      // This writes "now" and "!!"
    cerr << w << endl << s << endl;

    w = extractWord(s);
      // This writes "" and "" (both empty strings)
    cerr << w << endl << s << endl;
}
```

Hint: The string `substr` function may be of use in the implementation of this function.

bool isUppercase(string text)

This function returns true if and only if every letter in the string `text` is uppercase. (For a string with no letters, this means the function returns true.) Non-letter characters in the string `text`, if any, have no effect on the result of this function.

string makeUppercase(string text)

This function returns a new string identical to the value of the string `text`, except that each lowercase letter of `text` appears as its uppercase equivalent in the result string. For example, `makeUppercase("Earn *big* MONEY at home!!")` returns `"EARN *BIG* MONEY AT HOME!!"`.

bool hasMultipleExclamations(string text)

This function returns true if and only if the string `text` contains three consecutive exclamation marks. For example, it returns true for the strings `"Wow!!!"` and `"Congrats!!!! Good luck"`, and false for the strings `"W!I!N!"` and `"!! !"`.

bool isGibberishWord(string text)

This function returns true if and only if the string `text` contains *more than* three consecutive consonants. For example, it returns true for the strings `"AGPQrxab"` and `"xxxxozzzz"`, and false for the strings `"mortgage"` and `"discount prescriptions"`.

Programming Guidelines

The correctness of your program must not depend on undefined program behavior. Your program must never access out of range positions in a string.

You must use your best programming style, including commenting where appropriate. Be sure that your program builds successfully, and try to ensure that your functions do something reasonable for at least a few test cases. That way, you can get some partial credit for a solution that does not meet the entire specification. To test your program effectively, you should *unit test* each of your functions individually, and *system test* your entire program to ensure that all of the functions work together as expected.

One way to organize your program so that you can easily switch between testing individual functions and running the whole program normally is to do something like this:

```
const bool unitTesting = true;

int main()
{
    if (unitTesting)
```

```
    {
        doUnitsTests();
        return 0;
    }
    …  // code for the normal behavior goes here
}
```

The `doUnitTests` function does all your unit tests. To test the whole program normally, just change the value of the `unitTesting` variable from `true` to `false` (and make sure it's `false` when you turn in the program).

There are a number of ways you might write `doUnitTests`. One way is to interactively accept test strings:

```
void doUnitTests()
{
    string s;
    for (;;)
    {
        cerr << "Enter text: ";
        getline(cin, s);
        if (s == "quit")
            break;
        cerr << "isUppercase returns ";
        if (isUppercase(s))
            cerr << "true" << endl;
        else
            cerr << "false" << endl;
        cerr << "getFirstWord returns ";
        cerr << getFirstWord(s) << endl;
    }
}
```

While this is flexible, you run the risk of not being able to reproduce all your test cases if you make a change to your code and want to test that you didn't break anything that used to work.

Another way is to hard-code various tests and report which ones the program passes:

```
void doUnitTests()
{
    if (getFirstWord("hello there") == "hello")
        cerr << "Passed test 1: getFirstWord(\"hello there\") == \"hello\")" << endl;
    if (!isUppercase("WoW"))
```

```
        cerr << "Passed test 2: !isUppercase(\"WoW\")" << endl;
    …
}
```

This can get rather tedious. Fortunately, the library has a facility to make this easier: `assert`. If you include the header `<cassert>` you can call`assert` in the following manner:

```
assert(some boolean expression);
```

During execution, if the expression is true, nothing happens and execution continues normally; if it is false, a diagnostic message is written telling you the text and location of the failed assertion, and the program is terminated. As an example, here's a very incomplete set of tests:

```
#include <cassert>

…

void doUnitTests()
{
    assert(getFirstWord("hello there") == "hello");
    assert( isUppercase("WOW!!") );
    assert( !isUppercase("WoW!!") );
    string s = "***hello there";
    assert( extractWord(s) == "hello"  &&  s == " there" );
    assert( extractWord(s) == "there"  &&  s == "" );
    assert( extractWord(s) == ""  &&  s == "" );
    …
    cerr << "All tests succeeded" << endl;
}
```

The reason for writing one line of output at the end is to ensure that you can distinguish the situation of all tests succeeding from the case where one function you're testing silently crashes the program.

What to turn in

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

1. A text file named **spam.cpp** that contains the source code for your C++ program. Your source code should have helpful comments that tell the purpose of the major program segments and explain any tricky code.

Make sure that your program compiles and links successfully, and try to ensure that it does something reasonable for at least a few test cases. That way, you can get some partial credit for a program that does not meet the entire specification.

2. A file named **report.doc** or **report.docx** (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 usepseudocode in this description where it clarifies the presentation.
    c. A list of the test data that could be used to thoroughly test your program, along with the reason for each test. You don't have to include the results of the tests, but you must note which test cases your program does not handle correctly. (This could happen if you didn't have time to write a complete solution, or if you ran out of time while still debugging a supposedly complete solution.) Notice that most of this portion of your report can be written just after reading the requirements in this specification, before you even start designing your program.

By July 13, there will be a link on the class webpage that will enable you to turn in your zip file electronically. Turn in the file by the due time above. Give yourself enough time to be sure you can turn something in, because we will not accept excuses like "My network connection at home was down, and I didn't have a way to copy my files and bring them to a SEASnet machine." There's a lot to be said for turning in a preliminary version of your program and report early (You can always overwrite it with a later submission). That way you have something submitted in case there's a problem later.