Fall 2018 CS 31

Programming Assignment 3
House Party

Time due: 11:00 PM Tuesday, October 30

Introduction

You work for a polling company. Control of the House of Representatives is of major interest in the upcoming election, so polls are conducted in various states to predict how many of those states' congressional seats will go to each party. The data from the polls is encoded in a string that you will need to process. We will describe the format of the string and what you must do to process it.

First, we define some terms.

A *state code* is one of the following 50 two-letter codes, with each letter being in either upper or lower case (so CA Ca cA and ca are all state codes): AL AK AZ AR CA CO CT DE FL GA HI ID IL IN IA KS KY LA ME MD MA MI MN MS MO MT NE NV NH NJ NM NY NC ND OH OK OR PA RI SC SD TN TX UT VT VA WA WV WI WY

A *digit* is one of the ten digit characters `0` through `9`.

A *party code* is a letter, with upper and lower case versions being treated the same way. (The intent is that different letters represent different parties, e.g., D or d for Democratic, R or r for Republican, I or i for Independent, and other letters for various minor parties).

A *party result* is one or two digits immediately followed by a party code. For example, 14D and 6r and 01i and 0G are party results; 4 D is not, because of the space character between the 4 and the D. (The intent is that the digit(s) represent the number of seats that party is expected to win in some state.)

A *state forecast* is a state code immediately followed by zero or more party results. For example, CT5D and ne3r00D and NY9R17D1I and NJ3D5R4D and VT are state forecasts; KS 4R is not, because of the space character between the S and the 4. (The intent is that the party results represent the number of seats that various parties are expected to win in that state; a state forecast with no party results means the polls are so inconclusive that a prediction can not be made for even one seat.)

A *poll data string* is a sequence of zero or more state forecasts separated by commas (with no character that is not part of a state forecast in that string). No comma precedes the first state forecast or follows the last state forecast. For example, CT5D,NY9R17D1I,VT,ne3r00D is a poll data string consisting of four state forecasts; KS4R, NV3D1R is not a poll data string, because the space character between the comma and the N is not part of any state forecast. The empty string is a poll data string consisting of zero state forecasts.

These are the semantics of a poll data string: A poll data string represents a collection of predictions, one for each state forecast in that string. Each state forecast represents the prediction of the number of seats each indicated party in the listed party results will win in the state whose code is the state code; the digits in the party results represent those predicted numbers. For example, the poll string NY9R17D1I,VT,NJ3D5R4D,KS4R means that for the state NY, the predicted number of seats won will be 9 for party R, 17 for party D, and 1 for party I; for the state VT, no predictions; for the state NJ, 5 for party R and 7 for party D (3 for D and 4 more for D); for the state KS, 4 for party R.

Your task

Your assignment is essentially to take a poll data string and a party letter (in either upper or lower case), and compute the number of seats the poll data string predicts that party will win. For example, in the poll data string NY9R17D1I,Vt,NJ3d5r4D,KS4R the predicted number of seats for party D is 24 (17 from NY and 7 from NJ); for party R is 18 (9 from NY, 5 from NJ, and 4 from KS); and for party I is 1 (from NY).

For this project, you will implement the following two functions, using the exact function names, parameter types, and return types shown in this specification. (The parameter *names* may be different if you wish.)

```
bool hasProperSyntax(string pollData)
```

This function returns true if its parameter is a poll data string (i.e., it meets the definition above), or false otherwise.

```
int tallySeats(string pollData, char party, int& seatTally)
```

If the parameter `pollData` is not a poll data string, this function returns 1. If the parameter `party` is not a letter, this function returns 2. (If both of these situations occur, return one of those occurring situations' return values; it's your choice which one.) If either of the preceding situations occur, `seatTally` must be left unchanged. If neither of those situations occurs, then the function returns 0 after setting `seatTally` to the total number of seats that `pollData` predicts the party indicated by `party` will win.

These are the only two functions you are required to write. (Hint: `tallySeats` may well call `hasProperSyntax`.) Your solution may use functions in addition to these two if you wish. While we won't separately test additional functions you write, using them may help you structure your program more readably.

Of course, to test the functions you write, you'll want to write a main routine that calls your functions. During the course of developing your solution, you might change that main routine many times. As long as your main routine compiles correctly when you turn in your solution, it doesn't matter what it does, since we will rename it to something harmless and never call it (because we will supply our own main routine to thoroughly test your functions).

There are a couple of situations that would be important to deal with properly in the real world that we won't make you deal with to keep things simpler:

- If a poll data string contains two or more forecasts for the same state, `tallySeats` does not have to work correctly. We guarantee we will not test that function with poll data strings like IA3R,Ia1D or VA7R,KS4R,VA4D or VT,VT.
- If a state forecast in a poll data string indicates a total number of seats that is not the real world number of seats for the state, `tallySeats` must believe what the poll data string says. Thus, counting D seats in CA99D must yield 99, even though California really has only 53 seats in the House of Representatives.

Programming Guidelines

Your functions must not use any global variables whose values may be changed during execution (so global *constants* are allowed).

When you turn in your solution, neither of the two required functions, nor any functions you write 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.

The correctness of your program must not depend on undefined program behavior. Your program must never access out of range positions in a string. Your program must not, for example, assume anything about `n`'s value at the point indicated, or even whether or not the program crashes:

```
int main()
{
    string s = "Hello";
    int n;              // n is uninitialized
    s[5*n/n] = '!';     // undefined behavior!
    …
```

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 under both g31 and either Visual C++ or clang++. That way, you can get some partial credit for a solution that does not meet the entire specification.

If you wish, you may use this isValidUppercaseStateCode.txt function as part of your solution. (We can't imagine why you would not want to use it, since it does some of the work of validating a supposed state code.)

There are a number of ways you might write your main routine to test your functions. One way is to interactively accept test strings:

```
int main()
{
    for (;;)
    {
        cout << "Enter poll data string: ";
        string pds;
        getline(cin, pds);
        if (pds == "quit")
            break;
        cout << "hasProperSyntax returns ";
        if (hasProperSyntax(pds))
            cout << "true";
        else
            cout << "false";
        cout << 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:

```
int main()
{
    if (hasProperSyntax("CT5D,NY9R17D1I,VT,ne3r00D"))
        cout << "Passed test 1: hasProperSyntax(\"CT5D,NY9R17D1I,VT,ne3r00D")" << endl;
    if (!hasProperSyntax("ZT5D,NY9R17D1I,VT,ne3r00D"))
        cout << "Passed test 2: !hasProperSyntax(\"ZT5D,NY9R17D1I,VT,ne3r00D")" << endl;
    int seats;
    seats = -999;    // so we can detect whether tallySeats sets seats
    if (tallySeats("CT5D,NY9R17D1I,VT,ne3r00D", 'd', seats) == 0  &&  seats == 22)
        cout << "Passed test 3: tallySeats(\"CT5D,NY9R17D1I,VT,ne3r00D", 'd', seats)" << endl;
    seats = -999;    // so we can detect whether tallySeats sets seats
    if (tallySeats("CT5D,NY9R17D1I,VT,ne3r00D", '%', seats) == 2  &&  seats == -999)
        cout << "Passed test 4: tallySeats(\"CT5D,NY9R17D1I,VT,ne3r00D\", '%', seats)" << 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. Using `assert`, we can write the tests above more easily:

```
#include <iostream>
#include <cassert>
using namespace std;

bool hasProperSyntax(string pollData)
{
    … Your code goes here …
}

int tallySeats(string pollData, char party, int& seatTally)
{
    … Your code goes here …
}

int main()
{
    assert(hasProperSyntax("CT5D,NY9R17D1I,VT,ne3r00D"));
    assert(!hasProperSyntax("ZT5D,NY9R17D1I,VT,ne3r00D"));
    int seats;
    seats = -999;    // so we can detect whether tallySeats sets seats
    assert(tallySeats("CT5D,NY9R17D1I,VT,ne3r00D", 'd', seats) == 0  &&  seats == 22);
    seats = -999;    // so we can detect whether tallySeats sets seats
    assert(tallySeats("CT5D,NY9R17D1I,VT,ne3r00D", '%', seats) == 2  &&  seats == -999);
    …
    cout << "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 **poll.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. The file must be a complete C++ program that can be built and run, so it must contain appropriate #include lines, a main routine, and any additional functions you may have chosen to write.

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.
    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 October 29, 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.