Fall 2020 CS 31

Programming Assignment 3
Poll Position

Time due: 11:00 PM Tuesday, November 3

Part 1

Go through section 4 and section 5.1 of the zyBook. As usual, if you get to topics you're not already totally comfortable with, do most of the PAs and CAs. (This may well be the case with section 5.1 especially.) Do the following labs:

- 3.46
- 3.47
- 3.48
- 3.54
- 3.55
- 4.9
- 4.10
- 4.12

For this part, you won't turn anything in through the CS 31 web site; the zyBook system retains your lab submissions. It would be wise to do this part sooner rather than later.

Part 2

Introduction

You work for a polling company. At various times during a presidential campaign, polls are conducted in various states to predict which party's candidate will win that state's electoral votes. 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 letter is one of the 52 letter characters `A` through `Z` and `a` through `z`.

A state code is one of the following 51 (residents of the District of Columbia can vote) 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 DC 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, L or l for Libertarian, G or g for Green, and other letters for various minor parties).

A state forecast is exactly one or two digits immediately followed by a state code immediately followed by a party code. For example, 55CAD and 6msR are state forecasts; 4HI d is not, because of the space character between the I and the d. (The intent is that the digit(s) represent the number of electoral votes that state has.)

A poll data string is a sequence of zero or more state forecasts (with no character that is not part of a state forecast in that string). For example, 38TXR55CAD6Msr29nYd06UTL is a poll data string consisting of five state forecasts; neither 38TXR:55CAD nor 38TXR 55CAD is a poll data string, because neither the colon nor the space character after the R is 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 electoral votes the candidate of the party indicated by the party code will win in the state whose code is the state code; the digits in the state forecast represent that predicted number. For example, 38TXR represents a prediction that the candidate of party R will win 38 electoral votes from the state TX.

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 electoral votes the poll data string predicts the candidate of that party will get. For example, in the poll data string 38TXR55CAD6Msr29nYd06UTL, for the party D the predicted number of votes is 84 (55 from CA plus 29 from NY); for the party r it's 44 (38 from TX plus 6 from MS); for the party L, it's 6 (from UT).

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 isSyntacticallyCorrect(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 tallyVotes(string pollData, char party, int& voteTally)
```

> If the parameter `pollData` is not a poll data string (i.e., it does not meet the definition above), this function returns 1. If the parameter `party` is not a letter, this function returns 2. If `pollData` is a poll data string in which at least one state forecast predicts zero electoral votes for that state, this function returns 3. (If more than one of these situations occur, return one of those occurring situations' return value; it's your choice which one.) If any of the preceding situations occur, `voteTally` must be left unchanged. If none of those situations occurs, then the function returns 0 after setting `voteTally` to the total number of electoral votes that `pollData` predicts the candidate of the party indicated by `party` will get.

These are the only two functions you are required to write. (Hint: `tallyVotes` may well call `isSyntacticallyCorrect`.) Your solution may use functions in addition to these two if you wish. While we won't test those additional functions separately, 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, `tallyVotes` does not have to work correctly. We guarantee we will not test that function with poll data strings like 3MED1MeR or 52CAD38TXR3CAD.
- If a state forecast in a poll data string indicates a number of electoral votes that is not zero, but is not the real world number of electoral votes for the state, `tallyVotes` must believe what the poll data string says. Thus, tallying D votes in 99CAD must yield 99, even though California really has only 55 electoral votes.

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.at(5*n/n) = '!';  // undefined behavior!
    …
```

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 valididating a supposed state code.)

You do not need to know anything about arrays to write this program. You may use arrays if you wish, but the most straightforward solutions to this project actually don't use arrays.

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 << "isSyntacticallyCorrect returns ";
        if (isSyntacticallyCorrect(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 (isSyntacticallyCorrect("38TXR55CAD"))
        cout << "Passed test 1: isSyntacticallyCorrect(W"38TXR55CADW")" << endl;
    if (!isSyntacticallyCorrect("38MXR55CAD"))
        cout << "Passed test 2: !isSyntacticallyCorrect(W"38MXR55CADW")" << endl;
    int votes;
    votes = -999;    // so we can detect whether tallyVotes sets votes
    if (tallyVotes("38TXR55CAD6Msr29nYdO6UTL", 'd', votes) == 0  &&  votes == 84)
        cout << "Passed test 3: tallyVotes(W"38TXR55CAD6Msr29nYdO6UTL", 'd', votes)" << endl;
    votes = -999;    // so we can detect whether tallyVotes sets votes
    if (tallyVotes("38TXR55CAD", '%', votes) == 2  &&  votes == -999)
        cout << "Passed test 4: tallyVotes(W"38TXR55CADW", '%', votes)" << 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 isSyntacticallyCorrect(string pollData)
{
    … Your code goes here …
}

int tallyVotes(string pollData, char party, int& voteTally)
{
    … Your code goes here …
```

```
}

int main()
{
    assert(isSyntacticallyCorrect("38TXR55CAD"));
    assert(!isSyntacticallyCorrect("38MXR55CAD"));
    int votes;
    votes = -999;    // so we can detect whether tallyVotes sets votes
    assert(tallyVotes("38TXR55CAD6Msr29nYdO6UTL", 'd', votes) == 0  &&  votes == 84);
    votes = -999;    // so we can detect whether tallyVotes sets votes
    assert(tallyVotes("38TXR55CAD", '%', votes) == 2  &&  votes == -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 part 2 of 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 November 2, 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. 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.