Programming Assignment 3
Quality Control (QC) Testing

Time due: 9:00 PM ~~Tuesday, February 6th~~ Friday, February 9th.

Introduction

For this assignment, suppose that an automated manufacturing tracker records production QC test results in batches in a results string. For example,

Q2p1d1

would indicate that a single batch of two QC tests was performed with the results of one pass and one defect. Note that the character Q is used to start a batch of QC test results. The character p is to identify the number of tests that passed the QC test, and the character d indicates the number of defects. More than one batch of QC test results can be reported in a single results string. For example,

Q2d1p1Q5p3d2

would indicate two batches of QC test results, one with two test results and one with five test results, with a combined total of four passes and three defects.

Precisely, to be a valid QC test results string,

- a batch of results must begin with the character Q (case sensitive)
- a batch of results must report both pass and defect test results with p and d in either order
- no leading zeros are allowed in any numeric value being reported
- the total number of QC tests in a batch must equal the number of pass and defect test results.
- the total number of QC tests in a batch must be greater than zero (0).
- a single result string may include multiple batches of results

All of the following are examples of valid result strings:

- Q1p0d1Q1d0p1 (two batches of results, two total tests, one pass, one defect)
- Q5d2p3 (one batch of results, five total tests, two defects, three passes)

All of the following are examples of invalid result strings:

- q1p0d1 (batch must be reported with Q)
- Q1pd1 (a number for pass is required)
- Q1p1d (a number for defect is required)
- Q1p0d1 asdfR (extra characters not allowed)
- Q5p00003d0002 (leading zeros not allowed)

- Q5p0d0 (pass and defect results must equal the total number of tests)
- Q0p0d0 (batch cannot be zero)

Your task

For this project, you will implement the following five 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 isValidQC(string results)

This function returns true if its parameter is a well-formed test result string described above or false otherwise.

int passQC(string results)

If the parameter is a well-formed test result string, this function should return the total number of pass test results from all the batches reported in the string. If the parameter is not valid, return -1 .

int defectQC(string results)

If the parameter is a well-formed test result string, this function should return the total number of defect test results from all the batches reported in the string. If the parameter is not valid, return -1.

int totalQC(string results)

If the parameter is a well-formed test result string, this function should return the total number of tests being reported from all the batches in the string. If the parameter is not a valid, return -1.

int batches(string results)

If the parameter is a well-formed QC test result string, this function should return the total number of batches reported in the string. If the parameter is not a valid, return -1.

These are the only five functions you are required to write. Your solution may use functions in addition to these five 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 them, 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 test your functions thoroughly).

Clearly, I am expecting the last four functions to invoke the first function as they complete their assigned task. This code reuse is one of the major benefits of having functions. So please do that.

Programming Guidelines

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

When you turn in your solution, none of the five required functions, nor any functions 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.)

The correctness of your program must not depend on undefined program behavior. For example, you can assume nothing about c 's value at the point indicated, nor even whether or not the program crashes:

```
int main()
{
   string s = "Hello";
   char c = s[5];   // c's value is undefined
   …
```

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.

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

```
int main()
{
   string s;
cout.setf( ios::boolalpha ); // prints bool values as "true" or "false"
for(;;)
{
   cout << "Enter a possible teststring: ";
   getline(cin, s); if (s == "quit") break;
   cout << "isValidQC returns ";
            cout << isValidQC(s) << endl;
   cout << "pass test result(s) returns ";
   cout << passQC(s) << endl;
   cout << "defectQC(s) returns ";
   cout << defectQC(s) << endl;
   cout << "totalQC(s) returns ";
```

```
    cout << totalTests(s) << endl;
    cout <<< "batches(s) returns ";
    cout << batches(s) << endl;
      }
  return( 0 );
  }
```

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 (!isValidQC(""))
         cout << "test 1 OK: !isValidQC(\"\")" << endl;
   if (!isValidQC("   "))
         cout << "test 2 OK: !isValidQC(\"   \")" << endl;


  return( 0 );
}
```

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 to cerr 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>
#include <iostream>
#include <string>
using namespace std;


…
```

```
int main()
{
```

```
int main()
    {
        assert( isValidQC("") == false );
        assert( isValidQC("    ") == false );
        assert( passQC( "    " ) == -1 );
        assert( defectQC( "     " ) == -1 );
        assert( totalQC( "     " ) == -1 );
        assert( batches( "     " ) == -1 );
        assert( isValidQC( "Q2p1d1" ) == true );
        assert( passQC( "Q2p1d1" ) == 1 );
        assert( defectQC( "Q2p1d1" ) == 1 );
        assert( totalQC( "Q2p1d1" ) == 2 );
        assert( batches( "Q2+1-1" ) == -1 );

       cout << "All test cases succeeded" << endl;
    return( 0 );
     }
```

The reason for writing one line of output at the end is to ensure that you can distinguish the situation of all test cases 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 **qctest.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.doc** or **report.docx** (in Microsoft Word format) or **report.txt** (an ordinary text file) that contains in addition **your name** and **your UCLA Id Number** :

    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.)

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. Notice that most of the test data portion of your report can be written from the requirements in this specification, before you even start designing your program.

G31 Build Commands

```
g31 -c qctest.cpp
g31 qctest.o -o runnable
./runnable
```