CISE 2018
Programming Assignment 4
Functions

Time due: 12:00 PM, Friday, September 7 2018

Introduction

Elbosoft has decided to develop Heroic Grizblatnik, a music rhythm game suspiciously similar to one formerly distributed by Activision. In this game, a player must press buttons on a plastic stick in accordance with the sequence of symbols that appears on a screen. (That description sounds pretty dry; the entertainment comes from the fact that the plastic stick is in the shape of a grizblat (an Elbonian guitar-like instrument), a familiar Elbonian folk song is playing, the symbols appearing on the screen represent finger positions on the neck of a grizblat as a model of the notes to be played in time with the beat of the song, and the player can fantasize that he/she is actually talented enough to be up on stage. The symbols are five colored discs: green, red, yellow, blue, and orange. Here's an example of a similar game: https://www.youtube.com/watch?v=cHRfbiwdheg

A full Heroic Grizblatnik system has songs that require a player to press more than one button at the same time, simulating a chord. For this project, we will assume something simpler: All songs require pressing at most one button per beat. A press is either a *normal note* or a *sustained note*, where the player's finger presses the button on one beat and then stays there, being lifted off the button at a later beat.

We have acquired some Heroic Grizblatnik software that will display on a screen the sequence of colored discs for a song. The software requires as input a string of characters; each character is an instruction for what to display for one beat of the song. The characters are:

- `g`, `r`, `y`, `b`, or `o`, meaning to display a green, red, yellow, blue, or orange disc, representing a normal note.
- `G`, `R`, `Y`, `B`, or `O`, meaning to display a green, red, yellow, blue, or orange bar, representing a sustained note.
- `x`, meaning to display nothing for this beat, indicating that the player must not press any button for this beat.

(Notice that the instructions `gggb` and `GGGb` are different. The first means "press and release the green button for each of the first three beats, then press and release the blue button", while the second means "press the green button on the first beat, hold it there until you release it at the end of the third beat, then press and release the blue button".)

Now for the bad news: The game artists who encode the songs are used to expressing them a different way. For them, a song is expressed as a string like `r//y/g3///o/`, where a slash terminates every beat. This means "For the first beat, press and release red. Press nothing for the second beat. For the third beat, press and release yellow. Starting on the fourth beat, press and hold the green button three beats (the fourth, fifth, and sixth). For the seventh beat, press and release orange."

Your assignment is essentially to translate a song from the artists' representation to the sequence of instructions the Heroic Grizblatnik software wants. For example, the song `r//y/g3///o/` should be translated to the instructions `rxyGGGo`.

Let's define the syntax of the artists' song strings you are to translate. A *color* is one of these ten letters: `G g R r Y y B b O o`. A *digit* is one of the ten digit characters `0` through `9`. A *beat* is any one of the following:

- a slash (/)
- a color followed by a slash
- a color followed by a digit followed by a slash
- a color followed by two digits followed by a slash

A *syntactically correct song* is a sequence of zero or more beats. Every character in a non-empty syntactically correct song must be part of a beat (so, for example, `r/y` is not a syntactically correct song because the `y` is not part of a beat, since every beat must end with a slash). Here are some examples of syntactically correct songs:

- *zero beats*
- `G/`
- `r//Y/g3///o/`
- `y03///r10//////////`
- `///` *three beats*

If we are to successfully translate a syntactically correct song, it must meet a few additional contraints that go beyond its just having correct syntax. (This is akin to a sentence like "The orange truth ate moonbeams." being syntactically correct English, but meaningless, since it violates semantic constraints like "truth has no color" and "moonbeams can't be eaten".) In particular, although songs like `g3/r/y/` and `r5//` and `r0/` are syntactically correct, once we try to interpret their first beats as sustained notes, they violate our concept of what a sustained note should mean.

We therefore define a *translatable song* as one for which all of the following conditions hold:

- The song is syntactically correct.
- For all beats for which a sustained note is in effect, except the first, the beat consists only of a slash. (Thus, `o/r3///` is translatable, but `o/r3//y/` is not: Since we're restricted to playing one note at a time, then if we're sustaining a press of the red button, we can't also press the yellow button.)
- All beats for which a sustained note is in effect must be in the song; in other words, the song string can't end prematurely. (Thus, `o/r3///` is translatable, but `o/r3//` is not.)
- The length of a sustained note cannot be 0 or 1. (We could have decided differently, but we're decreeing this. Thus, `r03///` is translatable, but `r0/` and `r1/`are not.)

Here is how a translatable song is translated into instructions for the Heroic Grizblatnik software: Each beat will translate into one instruction letter. Although the case of an instruction letter matters in the instruction string (distinguishing a normal note from a sustained note), the case of a letter in the syntactically correct song string is irrelevant. Beats are translated as follows:

- If no sustained note is in effect, a beat consisting of only a slash is translated as `x`.
- If no sustained note is in effect, a beat consisting of a color followed by a slash is translated as the lower case version of the color letter.
- If a beat consists of a color followed by one or two digits followed by a slash, the digit(s) are interpreted as a decimal number denoting the number of beats a sustained note is to be in effect, starting with the current beat. Each beat for which a sustained note is in effect is translated as the upper case version of the color letter that started the sustained note.

(Notice that we do not define how a song that is not translatable is translated.) Here are some examples of how translatable songs are translated to instructions:

- The empty string translates to the empty string
- `//` translates to `xx`
- `g/G/g/B/` translates to `gggb`
- `g3///B/` translates to `GGGb`
- `r//y/g3///o/` translates to `rxyGGGo`

Your task

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 hasCorrectSyntax(string song)
```

This function returns true if its parameter is a syntactically correct song (i.e., it meets the definition above), and false otherwise.

```
int translateSong(string song, string& instructions, int& badBeat)
```

If the parameter `song` is translatable, the function sets `instructions` to the translation of the song, leaves `badBeat` unchanged, and returns 0. In all other cases, the function leaves `instructions` unchanged. If `song` is not syntactically correct, the function leaves `badBeat` unchanged and returns 1. If `song` is syntactically correct but a beat specifies a sustained note of length less than 2, `badBeat` is set to the number of that beat (where the first beat of the whole song is number 1, the second is number 2, etc.), and the function returns 2. If `song` is syntactically correct but while a sustained note is in effect, a beat not consisting of only a slash is present, `badBeat` is set to the number of that beat, and the function returns 3.

If `song` is syntactically correct but ends prematurely, `badBeat` is set to one more than the number of beats in the string, and the function returns 4.

If the song is syntactically correct but not translatable for more than one reason, report the leftmost occurring problem. (For example, if `song` is `r3//y/b0//o2/`, set badBeat to 3 and return 3.) In a case like `r3//b0/`, you must set `badBeat` to 3, but it's your choice whether you return 2 (length of a sustained note must be at least 2) or 3 (third beat must consist of only a slash), since neither error occurs earlier than the other.)

You must *not* assume that `instructions` and `badBeat` have any particular values at the time this function is entered.

These are the only two functions you are required to write. (Hint: `translateSong` may well call `hasCorrectSyntax`.) 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 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 thoroughly test your functions).

Programming Guidelines

The functions you write 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. For example, you can assume nothing about `n`'s value at the point indicated, nor even whether or not the program crashes:

```
int main()
{
    string s = "Hello";
    int n;              // n is uninitialized
    s[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. That way, you can get some partial credit for a solution that does not meet the entire specification.

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()
{
    string d;
    for (;;)
    {
        cout << "Enter song: ";
        getline(cin, d);
        if (d == "quit")
            break;
        cout << "hasCorrectSyntax returns ";
        if (hasCorrectSyntax(d))
            cout << "true" << endl;
        else
            cout << "false" << 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 (hasCorrectSyntax("g/b//"))
        cout << "Passed test 1: hasCorrectSyntax(\"g/b//\")" << endl;
    if (!hasCorrectSyntax("g/z//"))
        cout << "Passed test 2: !hasCorrectSyntax(\"g/z//\")" << 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 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()
{
    assert(hasCorrectSyntax("r/"));
    assert( ! hasCorrectSyntax("r"));
    string instrs;
    int badb;
    badb = -999;  // so we can detect whether this gets changed
    assert(translateSong("r//g/", instrs, badb) == 0  &&  instrs == "rxg"
&&  badb == -999);
    instrs = "WOW";  // so we can detect whether this gets changed
```

```
            badb = -999;  // so we can detect whether this gets changed
            assert(translateSong("r", instrs, badb) == 1  &&  instrs == "WOW"  &&
badb == -999);
            assert(translateSong("r/y3//g/r/", instrs, badb) == 3  &&  instrs ==
"WOW"  &&  badb == 4);
            …
            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 **hero.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** given below, 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**.**

Pseudocode

Pseudocode is usually a more effective means of communicating an algorithm than a narrative paragraph. Pseudocode should not be merely a statement-by-statement rephrasing of the code — how would that be any clearer than the code itself? For example, if we had to describe the algorithm for finding the average length of the words in a string as implemented by this code:

```
...
int totLength = 0;
int nWords = 0;
for (size_t pos = 0; ; )
{
```

```
            // find start of word
        while (pos != s.size()  &&  ! isalpha(s[pos]))
            pos++;

            // if no word, break
        if (pos == s.size())
            break;

        size_t start = pos;

            // find end of word
        do
        {
            pos++;
        } while (pos != s.size()  &&  isalpha(s[pos]));

        totLength += pos - start;
        nWords++;
    }
    if (nWords == 0)
        cout << "There are no words in the string" << endl;
    else
        cout << "The average word length is "
             << static_cast(totLength) / nWords << endl;
    ...
```

a suitable pseudocode rendition would be:

```
    ...
    repeatedly:
        find start of next word
        if no remaining words,
            break
        find end of word
        add word length to running total
        increment number of words
    write average length, or that there were no words
    ...
```

whereas the following is too detailed to give a clear understanding of what's going on, and just restates almost every line of code:

```
    ...
    set total length to 0
    set number of words to 0
    repeatedly:
        while current character is not alphabetic
            go on to next character
        if no remaining words,
            break
        save start position of the word
        while current character is alphabetic
            go on to next character
        add word length to total length
        increment number of words
    if there were no words,
        write the no word message
    else
        write the average length of the words
```

. . .

Presenting the algorithm as a sequence like the following doesn't make the loop structure of the algorithm visually clear:

1. Start a loop.
2. Find the start of the next word.
3. If there are no remaining words, break out of the loop.
4. Find the end of the word.
5. Add the word length to the running total.
6. Increment the number of words.
7. Loop back to step 2.
8. When the loop is done, write the average length, or that there were no words.

A narrative description like the following is practically useless; it's too detailed and completely hides the structure of the code:

First, set the total length to 0 and the number of words to 0. Then go into a loop. Inside the loop, first start a loop that checks every character until it finds a letter. If there was no letter, break out of the outer loop. Otherwise, save the start position of the word. Then start another loop that checks every character looking for a non-letter marking the end of the word. Add the length of the word to the total length and increment the number of words. When the outer loop is all done, if the number of words is 0, write the no word message, otherwise, write the average length.