Programming Assignment 3
Keyboard Koding

Time due: 9:00 PM Monday, October 30

Introduction

The site http://buttonbass.com/editor.html (requires Flash) features an online player piano. If you click on the `Notes` button just above the piano keyboard, and then `Computer Keyboard`, you will see that each key is labelled with a character. The text box just above it contains a sequence of these characters, indicating a song to be played on the piano. If you click `Play`, the sequence of notes will be played. If you clear the text box and type in or copy and paste a different sequence of characters, then the song that it represents will be played. The `Interval` control lets you specify the tempo, the number of milliseconds between the starts of each beat of the song.

The ButtonBass software expects the song to be represented as a sequence of instructions, where each instruction is either:

- one of the characters on the piano keyboard display: `ZXCVBNM,./ASDFGHJKLQWERTYUIOP1234567890!@#$%^&*()zxcvbnmasdfghjklqwertyuiop` Lower case letters represent the same piano key as their upper case equivalent.
- a space character, representing a rest that lasts one beat.
- an open square bracket `[`, followed by a sequence of characters on the piano keyboard display, followed by a close square bracket `]`. This represents a chord where all the indicated piano keys are played simultaneously for one beat.

(The ButtonBass software actually accepts a few more possibilities, but they are not relevant to this project.)

Now for the bad news: The people who want to produce songs want to use a notation that they find more natural, knowing the conventional way to name the piano keys. [Here's a video explanation of the conventional naming.] For them, a song is expressed as a string like `A3C#E//E//F#3A3D4/A3C#E/`, where a slash terminates every beat. This means "For the first beat, play a chord consisting of A in the third octave, and C# and E in the (default) fourth octave. For the second beat, play nothing. For the third beat, play E in the (default) fourth octave. Play nothing for the fourth beat. For the fifth beat, play a chord consisting of F# and A in the third octave, and D in the fourth octave. For the sixth beat, play a chord consisting of A in the third octave and C# and E in the (default) fourth octave."

Note: The Acoustical Society of America denotes the octave that ranges from middle C to the B above middle C as the fourth octave. The B just below middle C is the end of the third octave; the C one octave above middle C begins the fifth octave. The ButtonBass piano keyboard ranges from C2 (the C that starts the second octave) to C6 (the C that starts the sixth octave).

Your assignment is essentially to encode a song from the more natural representation to the sequence of characters the ButtonBass software wants. For example, the song `A3C#E//E//F#3A3D4/A3C#E/` should be encoded as the string `[D!J] J [8DH][D!J]`.

Let's define the syntax of the natural representation strings you are to encode.

A *note letter* is one of these seven letters, upper case only: `A B C D E F G`.

An *accidental sign* is one of these two characters: `# b`.

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

A *note* is either:

- a note letter
- a note letter immediately followed by an accidental sign
- a note letter immediately followed by a digit
- a note letter immediately followed by an accidental sign immediately followed by a digit

Thus the following are examples of notes: `D`, `Eb`, `F3`, and `C#5`.

A *beat* is a sequence of zero or more notes, immediately followed by a slash (`/`).

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

- *zero beats*
- `G/`
- `A3C#E//E//F#3A3D4/A3C#E/`
- `C0C0DC0DD/E#FbB#Cb/B#9/`
- `///` *three beats*

If ButtonBass is to successfully play a song string, that song string must meet an additional contraint that goes beyond its just being syntactically valid. (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, since the ButtonBass piano keyboard does not have any keys below C in the second octave or above C in the sixth octave, notes like Bb1 (intended to represent B-flat in the first octave) and F#8 (intended to represent F-sharp in the eighth octave) are not playable by the ButtonBass software.

We define a *playable note* as a note that can be played on the ButtonBass piano keyboard. (In what follows, we will abide by the customary convention that pairs like C# and Db are equivalent, that E# is equivalent to F in the same octave, as are E and Fb, that B# is equivalent to C in the next higher octave, and that Cb is equivalent to B in the next lower octave.) Thus, the playable notes are

- notes with no digit character. These are considered to be in octave 4, so D and D4 are equivalent, as are F# and F#4.

- notes with a digit character 2, 3, 4, or 5, except that Cb2 is not a playable note. These notes are considered to be in the indicated octave number.
- Cb6, representing Cb in the sixth octave, equivalent to B5.
- C6, representing C in the sixth octave.
- B#1, representing B# in the first octave, equivalent to C2.

A *playable song* is a song string that does not contain any notes that are not playable. Here is how a playable song is encoded as instructions for the ButtonBass software. Each beat will be encode as one instruction, so a song string will be encoded as a string consisting of a sequence of instructions. Beats are encoded as follows:

- A beat with no notes (consisting only of a slash) is encoded as a space character.
- A beat with exactly one note is encoded as a single character representing that note.
- A beat with more than one note is encoded as an open square bracket, immediately followed by the encodings of each of the notes in that beat, followed by a close square bracket.

(Notice that we do not define how a song that is not playable is encoded.) The encoding of a playable note is the single character that ButtonBass uses to represent that note. If that character is a letter, it must be the upper case version of that letter. For example, A4 is encoded as Q (not q), while Ab4 is encoded as $. Here are some examples of how playable songs are encoded:

- The empty string is encoded as the empty string
- `//` is encoded as two space characters
- `C/C/G/G/A/A/G/` is encoded as `GGLLQQL`
- `D3/F#3/A3/D4//D3F#3A3D4/` is encoded as `.8DH [.8DH]`
- `G3B3DD5//G/A/A3B/C5/B3D5//G//G//CE5//C5/D5/E5/F#5/B3G5//G//G/` is encoded as `[SFHR] LQ[DW]E[FR] L L [GT] ERT*[FU] L L`
- `DADDA/` is encoded as `[HQHHQ]`

The last one is unusual (Why have the same note appearing more than once in a chord?), but to keep things simple, we don't forbid it.

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 song string (i.e., it meets the definition above), and false otherwise.

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

If the parameter `song` is a playable song, the function sets `instructions` to the encoding of the song, leaves `badBeat` unchanged, and returns 0. In all other cases, the function leaves `instructions` unchanged. If `song` is not a song string, the function

leaves `badBeat` unchanged and returns 1. If `song` is a song string but is not playable, `badBeat` is set to the number of the beat with the first unplayable note (where the first beat of the whole song is number 1, the second is number 2, etc.), and the function returns 2. 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: `encodeSong` 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. 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, or 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.

If you wish, you may use this encodeNote function as part of your solution. (We can't imagine why you would not want to use it, since it does the work of converting one note to an instruction character, taking into account the octave and any accidental sign.)

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 t;
    for (;;)
    {
        cout << "Enter song: ";
        getline(cin, t);
        if (t == "quit")
            break;
        cout << "hasCorrectSyntax returns ";
        if (hasCorrectSyntax(t))
            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("D5//D/"))
        cout << "Passed test 1: hasCorrectSyntax(\"D5//D/\")" << endl;
    if (!hasCorrectSyntax("D5//Z/"))
        cout << "Passed test 2: !hasCorrectSyntax(\"D5//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>
using namespace std;

…

int main()
{
    assert(hasCorrectSyntax("D5//D/"));
    assert(!hasCorrectSyntax("D5//Z/"));
    string instrs;
    int badb;
    instrs = "xxx"; badb = -999; // so we can detect whether these get changed
    assert(encodeSong("D5//D/", instrs, badb) == 0  &&  instrs == "R H"  &&  badb
== -999);
```

```
            instrs = "xxx"; badb = -999; // so we can detect whether these get changed
            assert(encodeSong("D5//Z/", instrs, badb) == 1  &&  instrs == "xxx"  &&  badb
== -999);
            assert(encodeSong("D5//D8/", instrs, badb) == 2  &&  instrs == "xxx"  &&  badb
== 3);
            …
            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 **keyboard.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.